Mastering Modern C# for .NET

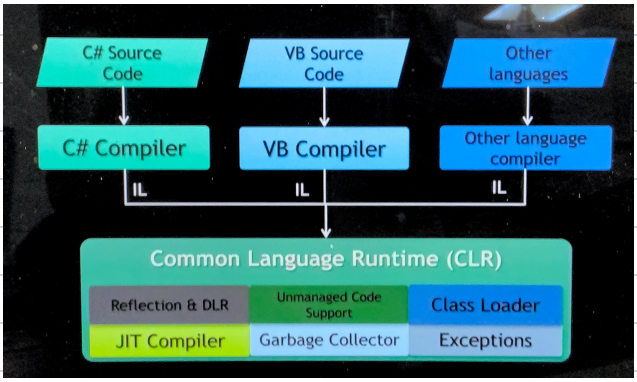
Lesson 1

Introduction to .NET and its Evolution

- Origins of .NET: .NET began as a Microsoft project in the late 1990s, initially as a response to Java, after Microsoft faced legal issues with Sun Microsystems over Java modifications. This led Microsoft to develop its own framework, which evolved into .NET.
- Mono Project: Originally a third-party initiative to make .NET applications cross-platform by running them on Linux, Mono was later acquired by Microsoft.
- Open Source and Cross-Platform Shift: With Microsoft's embrace of open-source and cross-platform compatibility, .NET was refactored into .NET Core, and eventually unified into a single platform simply called .NET.
- .NET Standard: This provides a common set of APIs to enable cross-platform compatibility across different versions of .NET, allowing developers to write code once and run it across multiple .NET runtimes.

Structure of .NET Applications

- Common Language Runtime (CLR): CLR is the virtual machine component of .NET, responsible for executing managed code. It handles memory management (garbage collection), type safety, exception handling, and more.
- Intermediate Language (IL): When .NET code is compiled, it's converted into an intermediate language (IL). The CLR then uses a Just-In-Time (JIT) compiler to translate IL into machine code at runtime.
- Assemblies and Metadata: Compiled .NET code is stored in assemblies, which contain both the IL code and metadata about the code. Metadata includes information about types, methods, and dependencies, allowing for features like reflection.
- Global Assembly Cache (GAC): Primarily used for storing assemblies shared among multiple applications, the GAC enables versioning and sharing of libraries within Windows. This is mainly relevant to the older .NET Framework.



.NET Runtime Variants

- .NET Core: A modern, modular version of .NET that supports cross-platform development.
- Mono Runtime: Allows .NET applications to run on Linux and macOS.
- AOT (Ahead of Time Compilation): A feature introduced to allow .NET applications to be pre-compiled for performance and size optimizations, eliminating the need for JIT compilation at runtime.
- RyuJIT: A JIT compiler with Profile-Guided Optimization (PGO), which re-compiles parts of code based on runtime behavior to improve performance.

Types of Class Libraries in .NET

In .NET, different types of class libraries can be created, each serving distinct purposes and offering compatibility with specific .NET runtimes and platforms. This flexibility allows developers to build libraries that are optimized for various use cases, whether targeting multiple platforms or leveraging specific platform features.

Platform-Specific Libraries

Platform-specific libraries are compiled and optimized for a specific runtime or platform. For example, a library might be compiled explicitly for the .NET Framework, .NET Core, or Xamarin.
Usage: This type of library is intended for applications that are limited to a specific environment, such as Windows-only applications (targeting .NET Framework) or mobile applications (targeting Xamarin).
Limitations: Since it is specific to one platform, it lacks cross-platform compatibility. You would need different versions of the library for different platforms if you want the code to run everywhere.

Portable Class Libraries (PCL)

PCLs were created as a way to write libraries that could run on multiple .NET platforms (e.g., .NET Framework, Silverlight, Windows Phone, and Xamarin).
Usage: PCLs allow a single library to target multiple platforms, but they were limited to a subset of APIs that were common across those platforms.
Limitations: The restricted API set meant that many platform-specific features weren't available in PCLs. Consequently, developers sometimes found PCLs too limited for complex applications. PCLs have since become obsolete and have largely been replaced by .NET Standard.

.NET Standard Libraries

.NET Standard is a formal specification of APIs that are available across all .NET implementations. By targeting a specific version of .NET Standard, a library can run on any .NET platform that supports that version.
Usage: .NET Standard libraries are designed for maximum compatibility across different .NET runtimes, including .NET Framework, .NET Core, Xamarin, and Mono. This allows developers to write a single library that can run across desktop, web, and mobile applications.
Versions: Different versions of .NET Standard specify different sets of APIs, with higher versions supporting more features but being compatible with fewer platforms. For example, .NET Standard 2.0 is compatible with .NET Framework 4.6.1 and later, while .NET Standard 2.1 is only compatible with .NET Core 3.x and .NET 5+.
Advantages: .NET Standard has largely replaced PCLs, offering a more extensive API surface and broader compatibility, making it the preferred choice for libraries that need to support multiple platforms.

GC.cpp File

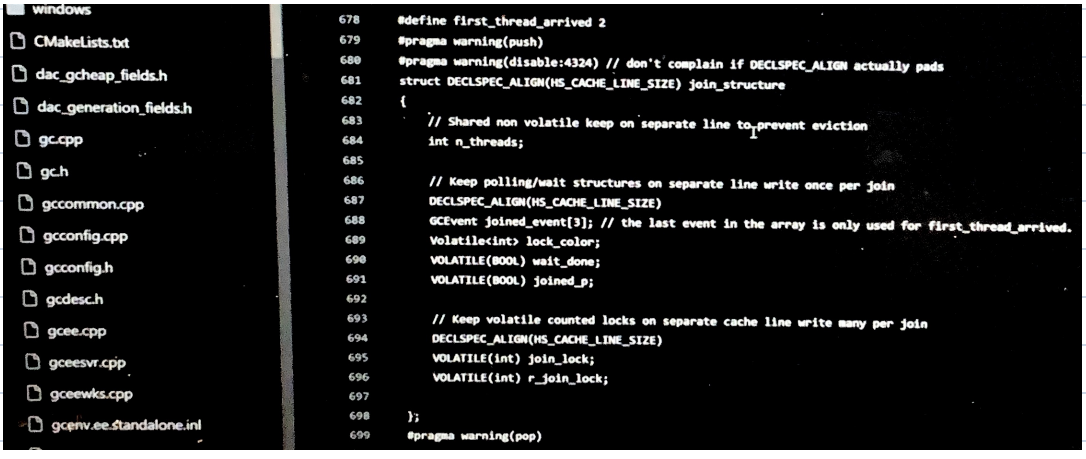
gc.cpp is a core C++ file within the .NET runtime that implements the garbage collection (GC) system in the Common Language Runtime (CLR). This file is critical to .NET's memory management and is exceptionally large, with over 53,000 lines of code. It handles tasks such as memory allocation, reclaiming unused memory, and optimizing the managed heap. These responsibilities include identifying objects no longer in use, performing collections, supporting different GC modes (workstation and server), and handling concurrent collections without stopping the entire application.

The complexity of gc.cpp arises from the need to manage various GC features through conditional compilation. This includes different compilation paths based on configuration settings, like whether the GC should run in server mode, workstation mode, or with background collection. The file's size and intricacy are partly due to its requirement to handle multiple GC scenarios and optimizations that adjust behavior according to application needs.

The primary maintainer of gc.cpp was deeply familiar with its contents, having written most of it. Instead of refactoring the file—which could introduce instability—she chose to document it extensively. This documentation helps others understand the highly specialized GC logic without risking changes to the performance-optimized and stable code.

Garbage collection in .NET is designed to be efficient and minimize application pauses. The GC in gc.cpp uses techniques such as generational collection, dividing objects into younger and older generations for more efficient memory management. The GC is also adaptable, supporting high-throughput scenarios and adjusting collection techniques based on the workload.

Open-sourcing .NET made gc.cpp accessible to the developer community, where it is now valuable both academically and practically as a resource for learning about garbage collection and memory management. Although there have been calls to refactor gc.cpp, documentation has been prioritized to preserve its stability and extensive optimizations.



Target Frameworks

- Started on .NET 5.0
- TFM (Target Framework Moniker) Simplifies the previous API targeting approach
- Instead of targeting .NET standard, you can target a specific .net runtime implementation
  - net 6.0, net 8.0, .net 9.0 → supported cross-platform runtimes
  - Net4.81 → Windows .NET Framework
  - Net8.0-windows → .NET 8 on Windows with WPF and WinForms
  - Net8.0-macos → .NET 8 on macOS
  - There are general targeting versions such as **net8.0-ios**, more and specific such as **net8.0-ios17.0**

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0-ios17.0</TargetFramework>
  </PropertyGroup>
  ...
</Project>
```

Manage Package Dependencies

- In the C# project file:
- The `<PackageReference>` project file element has the following structure:
  - `<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION"/>`
- To target conditional dependencies:
  - `<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" Condition="'$(TargetFramework)' == 'netcoreapp2.1'" />`
- You can add packages using Visual Studio, or via command line with: `dotnet add package <package name and version>`

Types of .NET applications deployments

Framework-Dependent Executables (FDE)

- Combines aspects of both FDD and SCD
- Produce a platform-specific executable that relies on a shared system-wide .NET runtime.
- The generated executable (e.g., app.exe on Windows) can run directly without using the **dotnet** command
- Advantages:
  - Smaller Size Compared to SCD
- Disadvantages:
  - Similar to FDD, the appropriate .NET runtime must be installed on the target system

Ahead-of-Time Compilation (AOT)

- AOT converts IL code into native machine code at build time
- Deployment Models:
  - Framework-Dependent AOT:
    - Relies on a shared system-wide .NET runtime, benefiting from AOT's performance enhancements
  - Self-Contained AOT:
    - Includes the .NET runtime
    - Produces a standalone native executable
- Pros:
  - Smaller memory footprint
  - Faster load and execution time
- Cons:
  - Increased Build Time
  - Larger Executables
  - Limited Dynamic Features

Types of .NET applications deployments

Framework-dependent deployment (FDD)

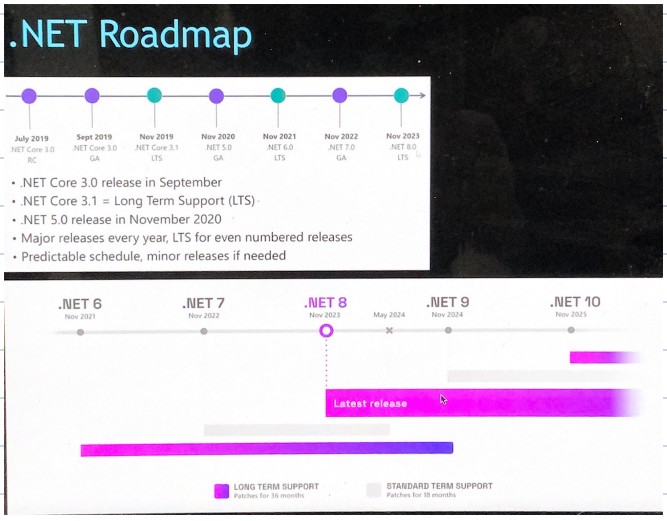
- Relies on the presence of a shared system-wide version of .NET on the target system.
- portable between installations of .NET Core.
- FDDs contain .dll files that can be launched by using the `dotnet` utility from the command line. For example, `dotnet app.dll`

Self-contained deployment (SCD)

- .NET libraries and Runtime, are included with the application
- isolated from other .NET applications.
- Include A renamed version of the platform-specific .NET host (such as app.exe on Window), and a .dll file (such as app.dll), which is the actual application.

.NET Framework Example Namespaces

Namespace	Description
System	Core classes and base classes for commonly used operations and services. E.g. string, math functions, garbage collection functions
System.Collections	Collection classes such as lists, queues and hashtables
System.IO	I/O related types dealing with files, directories, data streams, etc.
System.Data	Database access related types
System.Net	Network access related types
System.Reflection	Assembly and type metadata access
System.Diagnostics	Debugging, tracing, logging, etc.
System.Threading	Thread creation, synchronization and related
System.Xml	Manipulation of XML documents
System.Globalization	Support for localizing and globalizing applications

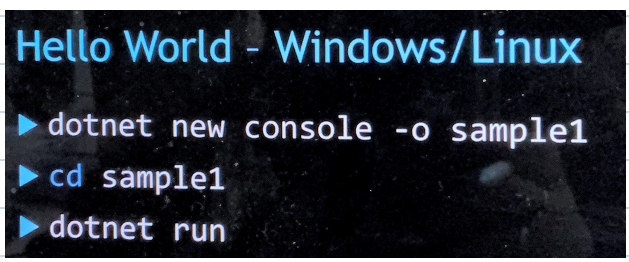


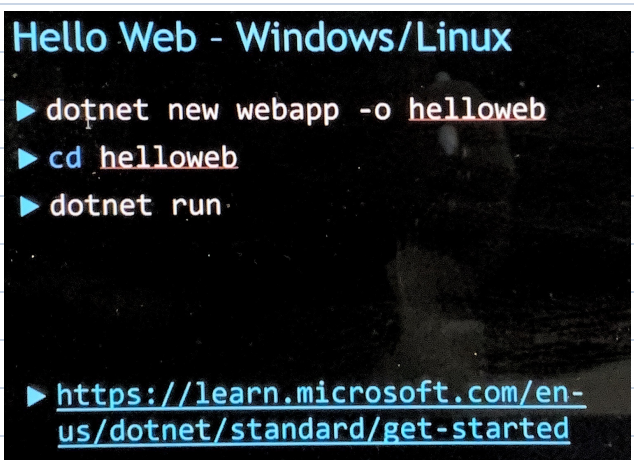
Release Cadence and Support Policies

- LTS (Long Term Support) vs. STS (Short Term Support): LTS versions of .NET are supported for three years, while STS versions are supported for only one year. LTS versions are released every two years.
- Version Compatibility: .NET tries to ensure backward compatibility, especially for long-term support versions, allowing developers to upgrade with minimal breaking changes.

Introduction to C# and Syntax Basics

- C# Standardization: C# is an ECMA standard language, meaning it has an open specification that allows other implementations.
- Syntax Similarity: C# syntax is similar to other C-style languages, like Java and C++, making it familiar for developers who have used these languages.
- Object-Oriented Language with Multi-Paradigm Support: C# is fully object-oriented but also supports functional, dynamic, and generic programming.
- Hello World in C#: Demonstrates the simplicity of C# syntax, especially in .NET 6 and later, where you can write a single line of code without needing a class or Main method.



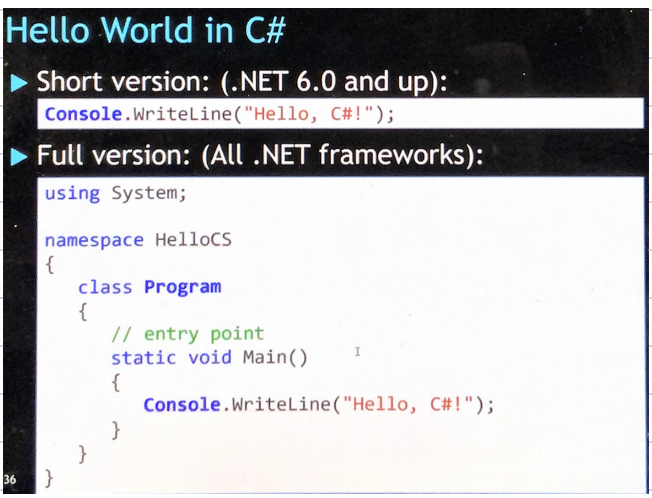


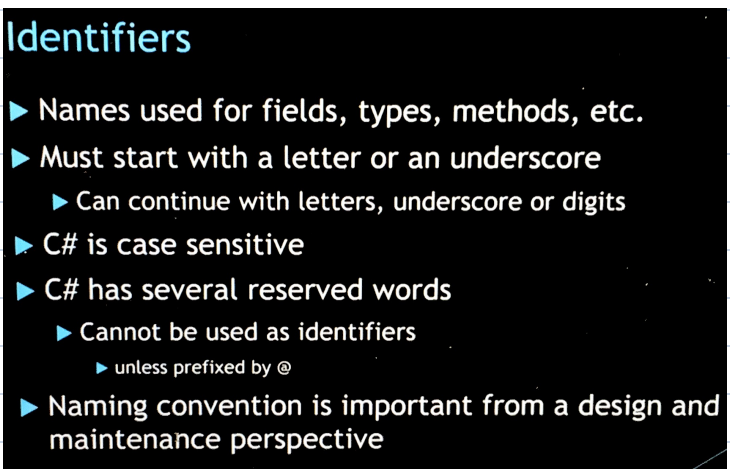
Development Tools for C#

- Visual Studio: The primary IDE for C#, offering powerful debugging and refactoring tools.
- Visual Studio Code: A lighter, open-source editor with C# support.
- Rider: A cross-platform IDE by JetBrains, favored for its speed and lightweight nature.

Types in C#

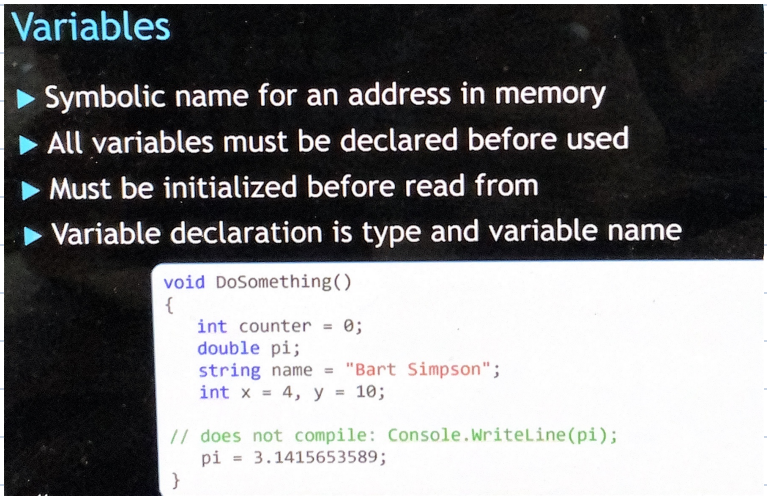
- Value Types and Reference Types: C# distinguishes between these two. Value types (like int, double) are stored on the stack, while reference types (like classes, objects) are stored on the heap.
- Built-in Types: The session discusses specific types (e.g., int, float, bool, decimal) and their use cases, including:
  - int for standard integer operations.
  - decimal for high-precision arithmetic, useful in financial calculations.
- Constants and Literals: Specifying the type of numeric literals using suffixes like L for long, F for float, and M for decimal.
- Unicode Support in Identifiers: C# allows Unicode characters in identifiers, allowing for non-ASCII names and underscores within numeric literals for readability (e.g., 1_000_000 for one million).

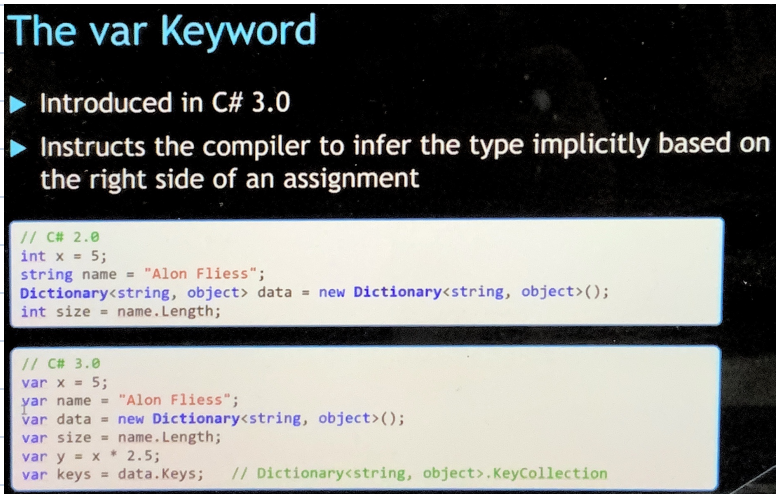




Variable Declarations and Type Inference

- var Keyword: Enables type inference by the compiler based on the assigned value. It's particularly useful in reducing code verbosity, especially with complex types.
- Default Initialization: C# initializes value types with default values (e.g., 0 for integers, false for booleans). Attributes like [SkipLocalsInit] can be used to disable this behavior in high-performance scenarios.





.NET / C# Integral Predefined Types

.NET type name	C# keyword	Size (bytes)	Description
System.Sbyte	sbyte	1	Signed byte (-128 to +127)
Systen.Byte	byte	1	Unsigned byte (0 to 255)
System.Int16	short	2	Signed 16-bit integer (-32768 to +32767)
System.UInt16	ushort	2	Unsigned 16-bit integer (0 to 65535)
System.Int32	int	4	Signed 32-bit integer
System.UInt32	uint	4	Unsigned 32-bit integer
System.Int64	long	8	Signed 64 bit integer
System.UInt64	ulong	8	Unsigned 64-bit integer
System.IntPtr	nint	4/8	Signed 32-bit or 64-bit (platform native)
System.UIntPtr	nuint	4/8	Unsigned 32-bit or 64-bit (platform native)
System.Char	char	2	A Unicode (UTF-16) character
System.Boolean	bool	1	A boolean value (false or true)

Skip Locals Init Attribute (C# >= 9.0)

- Instruct the compiler to not zero-out a variable upon instantiation
- *unsafe* operation!
- For Performance, only when you know what you're doing!

```
[SkipLocalsInit]
private void SomeMethod()
{
    byte[] buffer = new byte[100];  // Memory not zeroed out

    // Do something with the buffer
}
```

Numeric Literals

- Integer literals
  - Decimal or hexadecimal (0x prefix for hex)
  - Defaults to int, uint, long, ulong
  - L or l specifies long
  - U or u specifies unsigned

```
56      -65 7623u 0x56a
0x2000L     77L 123UL
```

- Floating point literals
  - Standard or scientific notation
  - double by default
  - F or f specifies float
  - D or d specifies double
  - M or m specifies decimal

```
4.0 3.2f    .67 3.88M
2.4e-9      1e8
6.66e-10
```

Other Literals

- Boolean
  - Can be true or false
- Single character
  - Single quotes
  - Character, escape (backslash), hex, Unicode

```
'a'     '\n'
'\x041'     '\u006f'
```

- String of characters
  - Double quotes
  - Can prefix with @ to disable escaping

```
"Hello, world!\n"
@"c:\temp\myfile.txt"
```

Binary Literals & Digits Separators (>= C# 7.0)

- `int x = 10;` ↔ `int x = 0b1010;`
- `int y = 33;` ↔ `int y = 0b100001;` ↔ `int y = 0b10_0001`
- `int z = 1048576;` ↔ `int z = 1_048_576;`

Default Literal Expressions (C# >= 7.1)

- Can omit the type on the right-hand side of the initialization

```
Func<string, bool> whereClause = default(Func<string, bool>); //old
Func<string, bool> whereClause = default; //new
int i = default(int); //old
int i = default; //new
```

Operators and Control Statements

- Basic Operators: Arithmetic (+, -, *, /), logical (&&, ||), bitwise (&, |, ^), and shift operators are discussed.
- Shift Operators: The addition of new shift operators in C# 11, including unsigned shifts, is introduced. This feature provides more control over bit manipulation, useful in low-level programming and performance optimizations.
- Pattern Matching: C# includes pattern matching features, such as is and is not, allowing more expressive code when working with types and data structures.
- Assignment Operators: Standard assignment (=) and compound assignments (e.g., +=, -=) are covered, highlighting common practices.

Arithmetic Operators

- Precedence
  - Multiplication, division, modulo
  - Addition, subtraction
- Can use parenthesis to change precedence

Operator	Description
+	Addition (positive value for unary)
-	Subtraction (negative value for unary)
*	Multiplication
/	Division
%	Modulo (integer remainder)

Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR
~	NOT
>>	Shift right
<<	Shift left
>>>	Unsigned right-shift (C# 11)

OR truth table

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

XOR truth table

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

AND truth table

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

```
Arithmetic Shift Right (ASR) of 1011 (4 bits, assuming two's complement):

1011 (which is -5 in two's complement) -> 1101 (preserves the sign, still -3)

Arithmetic Shift Left (ASL) of 1011 (4 bits):

1011 (which is -5) -> 0110 (which is +6)
```

Relational / Logical Operators

- Precedence
  - Not
  - And
  - Or
  - Relational operators
- Logical operators use "short circuit" evaluation

- Relational operators return true or false

Operator	Description
&&	And
||	Or
!	not

Operator	Description
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

```
bool r1 = x > 7;
bool r2 = y == 6 || x * 3 < 200;
bool r3 = z >= y && !y < 10 || x % 2 == 0;
```

55

Assignment Operators

Operator	Description
=	Simple assignment
op=	Compound assignment "op" is one of: + - * / % >> << & | ^
++	Increment
--	decrement

- Increment and decrement
  - Support postfix and prefix
  - Prefix means the value is updated first
  - Postfix means current value is used first

```
x = y + 3;      // simple assignment
z *= 2;         // compound
x = y++ * 3;    // value of y used before increment
z = 2 + ++y;    // y is incremented and its new value used
```

Arithmetic Operations

- Arithmetic operations use: `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`
  - Result type is the widest of operand types
  - `int` used if both operands are integral and narrower
  - Cannot mix `decimal` and `float`, or `ulong` with any signed type
- Otherwise, a cast may be needed (next slide)

```
byte + short      // widened to int
short + long      // widened to long
uint + short      // widened to long
ulong + int       // does not compile
short + decimal     // widened to decimal
```

Conversions and Casts

- Implicit cast silently convert to larger type
- Explicit cast is necessary when loss of information possible
- Casting is done by specifying the type to convert to in parenthesis

```
int x = 4;
long z = x;            // widening conversion
int k = z;             // does not compile
short s = (short)z;    // explicit cast - might lose data
```

Comments and Documentation in C#

- XML Documentation Comments: In addition to regular comments, C# supports XML documentation comments (///), which are used to generate inline documentation for classes and methods.
- Copilot and ChatGPT for Documentation: Tools like GitHub Copilot and ChatGPT can assist in generating documentation, which is helpful for better code readability and maintenance.

Statements, Expressions and Comments

- Statement
  - Comment, declaration, expression, control statement, block
- Expression
  - Anything that evaluates to a value
- Comments

```
// single line comment
```

```
/* this is a multiline comment
   can span as many lines as you like
*/
```

```
/// <summary>
/// this is XML comments
/// typically used for documentation
/// </summary>
```

42