C# Keywords

Complete Categorized Reference for C# 14 and C# 15 Preview

Verified against Microsoft documentation on July 30, 2026

77	46	3	4
Reserved	Stable contextual	C# 15 preview	Interop conventions

Scope and terminology

This reference contains every word in Microsoft's current C# keyword index, separates stable C# 14 terms from C# 15 preview additions, and includes the four case-sensitive calling-convention words recognized by the formal C# grammar. A reserved keyword cannot normally be used as an identifier. A contextual keyword has special meaning only in particular grammatical positions. Prefixing a keyword with @ can force it to be treated as an identifier where the grammar permits.

Recommended tagging model: store both a lexical class (reserved/contextual/preview) and a syntactic role (for example, object_creation, namespace_import, or property_accessor). The same keyword can perform more than one syntactic role.

Category map

- **Type declarations** (7 terms)
- **Accessibility and declaration modifiers** (18 terms)
- **Object, type, and member expressions** (8 terms)
- **Operators, conversions, and pattern logic** (8 terms)
- **Loops, jumps, returns, and iteration** (9 terms)
- **Parameters, generics, references, and constraints** (8 terms)
- **LINQ query expressions** (12 terms)
- **Type inference and top-level program context** (2 terms)
- **Built-in types and literal values** (23 terms)
- **Namespaces, imports, and aliases** (4 terms)
- **Properties and events** (8 terms)
- **Conditional branching and guards** (5 terms)
- **Exceptions, overflow, resources, and synchronization** (7 terms)
- **Asynchronous execution** (2 terms)
- **Native interop, unsafe code, and calling conventions** (9 terms)

1. Type declarations

Words that declare a new named type or type contract. **Suggested category tag:** `type_declaration`

Example: `public record Command(string Name);`

Keyword	Lexical class	Suggested tag	Purpose / function
`class`	Reserved	`type_declaration`	Declares a reference type whose instances can contain data and behavior.
`struct`	Reserved	`type_declaration`	Declares a value type, copied by value unless used through a reference.
`interface`	Reserved	`type_declaration`	Declares a contract of members that implementing types must provide.
`enum`	Reserved	`type_declaration`	Declares a value type made from a named set of integral constants.
`delegate`	Reserved	`type_declaration`	Declares a type-safe callable signature that can reference methods.
`record`	Contextual	`type_declaration`	Declares a data-oriented class or struct with generated value-based features.
`union`	**Preview keyword**	`type_declaration`	C# 15 preview: declares a type whose value is one of a fixed set of case types.

2. Built-in types and literal values

Language aliases for fundamental .NET types, plus built-in literal values. **Suggested category tag:** `built_in_type_or_literal`

Example: `int count = 0; string? name = null;`

Keyword	Lexical class	Suggested tag	Purpose / function
`bool`	Reserved	`built_in_type_or_literal`	Alias for System.Boolean; stores true or false.
`byte`	Reserved	`built_in_type_or_literal`	Alias for System.Byte; unsigned 8-bit integer.
`sbyte`	Reserved	`built_in_type_or_literal`	Alias for System.SByte; signed 8-bit integer.
`short`	Reserved	`built_in_type_or_literal`	Alias for System.Int16; signed 16-bit integer.
`ushort`	Reserved	`built_in_type_or_literal`	Alias for System.UInt16; unsigned 16-bit integer.
`int`	Reserved	`built_in_type_or_literal`	Alias for System.Int32; signed 32-bit integer.
`uint`	Reserved	`built_in_type_or_literal`	Alias for System.UInt32; unsigned 32-bit integer.
`long`	Reserved	`built_in_type_or_literal`	Alias for System.Int64; signed 64-bit integer.
`ulong`	Reserved	`built_in_type_or_literal`	Alias for System.UInt64; unsigned 64-bit integer.
`nint`	Contextual	`built_in_type_or_literal`	Contextual alias for System.IntPtr; signed native-sized integer.

Keyword	Lexical class	Suggested tag	Purpose / function
nuint	Contextual	built_in_type_or_literal	Contextual alias for System.UIntPtr; unsigned native-sized integer.
char	Reserved	built_in_type_or_literal	Alias for System.Char; one UTF-16 code unit.
float	Reserved	built_in_type_or_literal	Alias for System.Single; 32-bit floating-point value.
double	Reserved	built_in_type_or_literal	Alias for System.Double; 64-bit floating-point value.
decimal	Reserved	built_in_type_or_literal	Alias for System.Decimal; high-precision base-10 numeric value.
string	Reserved	built_in_type_or_literal	Alias for System.String; an immutable sequence of UTF-16 characters.
object	Reserved	built_in_type_or_literal	Alias for System.Object; the ultimate base type of .NET types.
dynamic	Contextual	built_in_type_or_literal	Defers member binding and many type checks until runtime.
void	Reserved	built_in_type_or_literal	Indicates that a method returns no value, or represents an untyped pointer target.
true	Reserved	built_in_type_or_literal	Boolean literal representing the true value.
false	Reserved	built_in_type_or_literal	Boolean literal representing the false value.
null	Reserved	built_in_type_or_literal	Literal representing no object reference or no nullable value.
default	Reserved	built_in_type_or_literal	Produces the default value of a type; also labels a fallback switch case.

3. Accessibility and declaration modifiers

Words that alter visibility, inheritance, storage, mutability, or declaration behavior. **Suggested category tag:** `declaration_modifier`

Example: `public static partial class CommandCatalog { }`

Keyword	Lexical class	Suggested tag	Purpose / function
public	Reserved	declaration_modifier	Makes a declared member or type accessible wherever its containing assembly is referenced.
private	Reserved	declaration_modifier	Restricts access to the containing type.
protected	Reserved	declaration_modifier	Allows access from the containing type and derived types.
internal	Reserved	declaration_modifier	Restricts access to the same assembly.

Keyword	Lexical class	Suggested tag	Purpose / function
abstract	Reserved	declaration_modifier	Declares an incomplete type or member that must be completed by derivation or override.
sealed	Reserved	declaration_modifier	Prevents a class from being inherited or an override from being overridden again.
static	Reserved	declaration_modifier	Makes a member belong to the type itself, or declares a non-instantiable static class.
virtual	Reserved	declaration_modifier	Allows an instance member to be overridden in a derived type.
override	Reserved	declaration_modifier	Provides a new implementation of an inherited virtual or abstract member.
readonly	Reserved	declaration_modifier	Restricts mutation of fields, structs, members, or references depending on context.
volatile	Reserved	declaration_modifier	Applies special memory-access semantics to a field; it is not a replacement for locking.
const	Reserved	declaration_modifier	Declares a compile-time constant whose value is embedded in consuming code.
partial	Contextual	declaration_modifier	Allows a type or supported member declaration to be split across multiple parts.
required	Contextual	declaration_modifier	Requires callers to initialize a field or property during object creation.
file	Contextual	declaration_modifier	Limits a top-level type to the source file in which it is declared.
extension	Contextual	declaration_modifier	C# 14: introduces an extension block containing extension members for another type.
closed	**Preview contextual**	declaration_modifier	C# 15 preview: restricts direct derived types to the declaring assembly for exhaustiveness analysis.
safe	**Preview contextual**	declaration_modifier	C# 15 preview: explicitly attests that a declaration is safe under the updated memory-safety model.

4. Namespaces, imports, and aliases

Words that organize names, import namespaces or static members, and create aliases. **Suggested category tag:** `namespace_or_import`

Example: `using IO = System.IO; namespace HyperShell.Commands;`

Keyword	Lexical class	Suggested tag	Purpose / function
`namespace`	Reserved	`namespace_or_import`	Declares the logical namespace that contains types or nested namespaces.
`using`	Reserved	`namespace_or_import`	Introduces namespace imports, aliases, static imports, or deterministic resource disposal.
`global`	Contextual	`namespace_or_import`	Refers to the global namespace or declares a project-wide global using directive.
`alias`	Contextual	`namespace_or_import`	Used with extern alias to reference an externally assigned assembly alias.

5. Object, type, and member expressions

Words used to create objects, refer to the current/base instance, inspect types, or construct values. **Suggested category tag:** `object_or_type_expression`

Example: `var copy = original with { Name = "New" };`

Keyword	Lexical class	Suggested tag	Purpose / function
`new`	Reserved	`object_or_type_expression`	Creates an object or value; can also hide an inherited member or specify a constructor constraint.
`this`	Reserved	`object_or_type_expression`	Refers to the current instance; also marks an extension-method receiver or indexer parameter list.
`base`	Reserved	`object_or_type_expression`	Accesses base-class members or invokes a base-class constructor.
`typeof`	Reserved	`object_or_type_expression`	Produces the System.Type object for a type known at compile time.
`sizeof`	Reserved	`object_or_type_expression`	Returns the size in bytes of an unmanaged type.
`nameof`	Contextual	`object_or_type_expression`	Produces a compile-time string containing the name of a symbol.
`stackalloc`	Reserved	`object_or_type_expression`	Allocates a block of memory on the stack, commonly exposed as a Span<T> or pointer.
`with`	Contextual	`object_or_type_expression`	Creates a modified copy or, in C# 15 collection expressions, supplies construction arguments.

6. Properties and events

Words that declare events and define property, indexer, and custom-event accessors. **Suggested category tag:** `property_or_event_keyword`

Example: `public string Name { get; init; }`

Keyword	Lexical class	Suggested tag	Purpose / function
`event`	Reserved	`property_or_event_keyword`	Declares an event member with controlled subscription and unsubscription.
`get`	Contextual	`property_or_event_keyword`	Defines the read accessor of a property or indexer.
`set`	Contextual	`property_or_event_keyword`	Defines the write accessor of a property or indexer.
`init`	Contextual	`property_or_event_keyword`	Defines a write accessor usable only during object initialization.
`add`	Contextual	`property_or_event_keyword`	Defines the accessor executed when code subscribes to a custom event.
`remove`	Contextual	`property_or_event_keyword`	Defines the accessor executed when code unsubscribes from a custom event.
`value`	Contextual	`property_or_event_keyword`	Implicit parameter containing the value supplied to a setter or custom event accessor.
`field`	Contextual	`property_or_event_keyword`	C# 14: refers to a compiler-generated backing field inside a property accessor.

7. Operators, conversions, and pattern logic

Words used to declare operators, define conversions, test types, cast, and combine patterns. **Suggested category tag:** `operator_conversion_or_pattern`

Example: `if (value is int and not 0) { }`

Keyword	Lexical class	Suggested tag	Purpose / function
`operator`	Reserved	`operator_conversion_or_pattern`	Declares a user-defined operator or conversion operator.
`implicit`	Reserved	`operator_conversion_or_pattern`	Declares a conversion that the compiler may apply without an explicit cast.
`explicit`	Reserved	`operator_conversion_or_pattern`	Declares a conversion that normally requires an explicit cast.
`is`	Reserved	`operator_conversion_or_pattern`	Tests a value against a type or pattern and can introduce a pattern variable.
`as`	Reserved	`operator_conversion_or_pattern`	Attempts a reference or nullable conversion and returns null when it fails.
`and`	Contextual	`operator_conversion_or_pattern`	Combines two patterns that must both match.

Keyword	Lexical class	Suggested tag	Purpose / function
or	Contextual	operator_conversion_or_pattern	Combines alternative patterns where either may match.
not	Contextual	operator_conversion_or_pattern	Negates a pattern.

8. Conditional branching and guards

Words that select among execution paths and add conditions to cases or exception handlers. **Suggested category tag:** conditional_control_flow

Example: `switch (state) { case Ready when enabled: break; }`

Keyword	Lexical class	Suggested tag	Purpose / function
if	Reserved	conditional_control_flow	Executes a statement when a Boolean condition is true.
else	Reserved	conditional_control_flow	Provides the alternative branch of an if statement.
switch	Reserved	conditional_control_flow	Selects a branch or value by matching cases or patterns.
case	Reserved	conditional_control_flow	Introduces a switch label or pattern arm.
when	Contextual	conditional_control_flow	Adds a filter to a catch clause or a guard to a switch case or arm.

9. Loops, jumps, returns, and iteration

Words that repeat code, transfer control, return from members, or produce iterator values. **Suggested category tag:** loop_or_jump_control_flow

Example: `foreach (var item in items) yield return item;`

Keyword	Lexical class	Suggested tag	Purpose / function
for	Reserved	loop_or_jump_control_flow	Defines a loop with initializer, condition, and iterator components.
foreach	Reserved	loop_or_jump_control_flow	Iterates over elements produced by an enumerable pattern.
while	Reserved	loop_or_jump_control_flow	Repeats a statement while a condition remains true.
do	Reserved	loop_or_jump_control_flow	Starts a loop that evaluates its condition after each iteration.
break	Reserved	loop_or_jump_control_flow	Exits the nearest loop or switch; C# 15 preview also supports labeled targets.

Keyword	Lexical class	Suggested tag	Purpose / function
`continue`	Reserved	`loop_or_jump_control_flow`	Starts the next loop iteration; C# 15 preview also supports labeled targets.
`goto`	Reserved	`loop_or_jump_control_flow`	Transfers control to a label, case label, or default label.
`return`	Reserved	`loop_or_jump_control_flow`	Exits a method, local function, lambda, or accessor and optionally supplies a value.
`yield`	Contextual	`loop_or_jump_control_flow`	Produces the next value from an iterator or ends iterator production.

10. Exceptions, overflow, resources, and synchronization

Words that handle exceptions, select overflow behavior, and coordinate synchronized execution. **Suggested category tag:** `runtime_control_or_safety`

Example: `try { checked { total += value; } } catch { throw; }`

Keyword	Lexical class	Suggested tag	Purpose / function
`try`	Reserved	`runtime_control_or_safety`	Introduces code whose exceptions may be handled or finalized.
`catch`	Reserved	`runtime_control_or_safety`	Handles an exception thrown from an associated try block.
`finally`	Reserved	`runtime_control_or_safety`	Runs cleanup code after try/catch completion regardless of success or exception.
`throw`	Reserved	`runtime_control_or_safety`	Raises an exception or rethrows the current exception.
`checked`	Reserved	`runtime_control_or_safety`	Enables overflow checking for integral arithmetic and conversions in a scope or expression.
`unchecked`	Reserved	`runtime_control_or_safety`	Disables overflow checking for integral arithmetic and conversions in a scope or expression.
`lock`	Reserved	`runtime_control_or_safety`	Executes a critical section with mutual exclusion on a synchronization object.

11. Parameters, generics, references, and constraints

Words that define parameter passing, by-reference behavior, generic constraints, and ref-safety limits. **Suggested category tag:** `parameter_reference_or_constraint`

Example: `void Copy<T>(in T source, out T result) where T : notnull`

Keyword	Lexical class	Suggested tag	Purpose / function
`params`	Reserved	`parameter_reference_or_constraint`	Allows a method parameter to accept a variable number of arguments or a parameter collection.
`in`	Reserved	`parameter_reference_or_constraint`	Marks a readonly by-reference parameter; also appears in foreach and query syntax.
`out`	Reserved	`parameter_reference_or_constraint`	Marks a by-reference output parameter that the callee must assign.
`ref`	Reserved	`parameter_reference_or_constraint`	Passes, returns, or stores an alias to an existing variable or storage location.
`where`	Contextual	`parameter_reference_or_constraint`	Introduces generic constraints or filters elements in a LINQ query.
`notnull`	Contextual	`parameter_reference_or_constraint`	Constrains a generic type parameter to non-nullable types under nullable analysis.
`allows`	Contextual	`parameter_reference_or_constraint`	Introduces an anti-constraint, currently used as allows ref struct for ref-like type arguments.
`scoped`	Contextual	`parameter_reference_or_constraint`	Restricts the escape lifetime of a ref-like parameter, local, or return value.

12. Asynchronous execution

Words that declare asynchronous code and suspend it until an awaitable operation completes. **Suggested category tag:** `asynchronous_keyword`

Example: `async Task RunAsync() => await Task.Delay(100);`

Keyword	Lexical class	Suggested tag	Purpose / function
`async`	Contextual	`asynchronous_keyword`	Marks a method, lambda, or anonymous function that can use await.
`await`	Contextual	`asynchronous_keyword`	Suspends asynchronous execution until an awaitable operation completes without blocking the thread.

13. LINQ query expressions

Contextual words that form SQL-like LINQ query-expression clauses. **Suggested category tag:** `query_expression_keyword`

Example: `from x in items where x.Enabled orderby x.Name select x;`

Keyword	Lexical class	Suggested tag	Purpose / function
`from`	Contextual	`query_expression_keyword`	Introduces a query data source and range variable.
`select`	Contextual	`query_expression_keyword`	Projects query elements into the result sequence.
`group`	Contextual	`query_expression_keyword`	Groups query elements by a key.
`into`	Contextual	`query_expression_keyword`	Continues a query using the result of a group, join, or select clause.
`orderby`	Contextual	`query_expression_keyword`	Sorts query results by one or more keys.
`join`	Contextual	`query_expression_keyword`	Correlates elements from two data sources by matching keys.
`let`	Contextual	`query_expression_keyword`	Introduces a query range variable for a computed intermediate value.
`on`	Contextual	`query_expression_keyword`	Introduces the left key expression of a query join clause.
`equals`	Contextual	`query_expression_keyword`	Separates the left and right key expressions of a query join clause.
`by`	Contextual	`query_expression_keyword`	Introduces the key expression of a group clause.
`ascending`	Contextual	`query_expression_keyword`	Requests ascending ordering for a query sort key.
`descending`	Contextual	`query_expression_keyword`	Requests descending ordering for a query sort key.

14. Native interop, unsafe code, and calling conventions

Words used for externally implemented members, pointer operations, pinning, function pointers, and native calling conventions. **Suggested category tag:** `interop_or_unsafe_keyword`

Example: `delegate* unmanaged[Cdecl]<int, int> callback;`

Keyword	Lexical class	Suggested tag	Purpose / function
`extern`	Reserved	`interop_or_unsafe_keyword`	Declares an externally implemented member or introduces an external assembly alias.
`unsafe`	Reserved	`interop_or_unsafe_keyword`	Permits pointer operations and other code requiring an unsafe context.

Keyword	Lexical class	Suggested tag	Purpose / function
`fixed`	Reserved	`interop_or_unsafe_keyword`	Pins managed storage for pointer access or declares a fixed-size buffer.
`managed`	Contextual	`interop_or_unsafe_keyword`	Specifies a managed function-pointer calling convention.
`unmanaged`	Contextual	`interop_or_unsafe_keyword`	Specifies an unmanaged function-pointer convention or constrains T to unmanaged types.
`Cdecl`	Contextual (interop)	`interop_or_unsafe_keyword`	Selects the C declaration unmanaged function-pointer calling convention.
`Fastcall`	Contextual (interop)	`interop_or_unsafe_keyword`	Selects the Fastcall unmanaged function-pointer calling convention.
`Stdcall`	Contextual (interop)	`interop_or_unsafe_keyword`	Selects the Standard Call unmanaged function-pointer calling convention.
`Thiscall`	Contextual (interop)	`interop_or_unsafe_keyword`	Selects the Thiscall unmanaged function-pointer calling convention.

15. Type inference and top-level program context

Contextual words whose meaning is supplied by the compiler in local declarations or top-level programs. **Suggested category tag:** `compiler_context_keyword`

Example: `var firstArgument = args[0];`

Keyword	Lexical class	Suggested tag	Purpose / function
`var`	Contextual	`compiler_context_keyword`	Requests compiler inference for a local variable type or captures a value in a var pattern.
`args`	Contextual	`compiler_context_keyword`	Names the compiler-provided string[] command-line argument variable in top-level statements.

Alphabetical inventory

All 130 terms, alphabetized. Preview terms are marked with an asterisk (*).

`abstract` Accessibility and declaration modifiers	**`add`** Properties and events	**`alias`** Namespaces, imports, and aliases	**`allows`** Parameters, generics, references, and constraints
`and` Operators, conversions, and pattern logic	**`args`** Type inference and top-level program context	**`as`** Operators, conversions, and pattern logic	**`ascending`** LINQ query expressions

async Asynchronous execution	**await** Asynchronous execution	**base** Object, type, and member expressions	**bool** Built-in types and literal values
break Loops, jumps, returns, and iteration	**by** LINQ query expressions	**byte** Built-in types and literal values	**case** Conditional branching and guards
catch Exceptions, overflow, resources, and synchronization	**Cdecl** Native interop, unsafe code, and calling conventions	**char** Built-in types and literal values	**checked** Exceptions, overflow, resources, and synchronization
class Type declarations	**closed*** Accessibility and declaration modifiers	**const** Accessibility and declaration modifiers	**continue** Loops, jumps, returns, and iteration
decimal Built-in types and literal values	**default** Built-in types and literal values	**delegate** Type declarations	**descending** LINQ query expressions
do Loops, jumps, returns, and iteration	**double** Built-in types and literal values	**dynamic** Built-in types and literal values	**else** Conditional branching and guards
enum Type declarations	**equals** LINQ query expressions	**event** Properties and events	**explicit** Operators, conversions, and pattern logic
extension Accessibility and declaration modifiers	**extern** Native interop, unsafe code, and calling conventions	**false** Built-in types and literal values	**Fastcall** Native interop, unsafe code, and calling conventions
field Properties and events	**file** Accessibility and declaration modifiers	**finally** Exceptions, overflow, resources, and synchronization	**fixed** Native interop, unsafe code, and calling conventions
float Built-in types and literal values	**for** Loops, jumps, returns, and iteration	**foreach** Loops, jumps, returns, and iteration	**from** LINQ query expressions
get Properties and events	**global** Namespaces, imports, and aliases	**goto** Loops, jumps, returns, and iteration	**group** LINQ query expressions
if Conditional branching and guards	**implicit** Operators, conversions, and pattern logic	**in** Parameters, generics, references, and constraints	**init** Properties and events
int Built-in types and literal values	**interface** Type declarations	**internal** Accessibility and declaration modifiers	**into** LINQ query expressions
is Operators, conversions, and pattern logic	**join** LINQ query expressions	**let** LINQ query expressions	**lock** Exceptions, overflow, resources, and synchronization
long Built-in types and literal values	**managed** Native interop, unsafe code, and calling conventions	**nameof** Object, type, and member expressions	**namespace** Namespaces, imports, and aliases
new Object, type, and member expressions	**nint** Built-in types and literal values	**not** Operators, conversions, and pattern logic	**notnull** Parameters, generics, references, and constraints

nuint Built-in types and literal values	**null** Built-in types and literal values	**object** Built-in types and literal values	**on** LINQ query expressions
operator Operators, conversions, and pattern logic	**or** Operators, conversions, and pattern logic	**orderby** LINQ query expressions	**out** Parameters, generics, references, and constraints
override Accessibility and declaration modifiers	**params** Parameters, generics, references, and constraints	**partial** Accessibility and declaration modifiers	**private** Accessibility and declaration modifiers
protected Accessibility and declaration modifiers	**public** Accessibility and declaration modifiers	**readonly** Accessibility and declaration modifiers	**record** Type declarations
ref Parameters, generics, references, and constraints	**remove** Properties and events	**required** Accessibility and declaration modifiers	**return** Loops, jumps, returns, and iteration
safe* Accessibility and declaration modifiers	**sbyte** Built-in types and literal values	**scoped** Parameters, generics, references, and constraints	**sealed** Accessibility and declaration modifiers
select LINQ query expressions	**set** Properties and events	**short** Built-in types and literal values	**sizeof** Object, type, and member expressions
stackalloc Object, type, and member expressions	**static** Accessibility and declaration modifiers	**Stdcall** Native interop, unsafe code, and calling conventions	**string** Built-in types and literal values
struct Type declarations	**switch** Conditional branching and guards	**this** Object, type, and member expressions	**Thiscall** Native interop, unsafe code, and calling conventions
throw Exceptions, overflow, resources, and synchronization	**true** Built-in types and literal values	**try** Exceptions, overflow, resources, and synchronization	**typeof** Object, type, and member expressions
uint Built-in types and literal values	**ulong** Built-in types and literal values	**unchecked** Exceptions, overflow, resources, and synchronization	**union*** Type declarations
unmanaged Native interop, unsafe code, and calling conventions	**unsafe** Native interop, unsafe code, and calling conventions	**ushort** Built-in types and literal values	**using** Namespaces, imports, and aliases
value Properties and events	**var** Type inference and top-level program context	**virtual** Accessibility and declaration modifiers	**void** Built-in types and literal values
volatile Accessibility and declaration modifiers	**when** Conditional branching and guards	**where** Parameters, generics, references, and constraints	**while** Loops, jumps, returns, and iteration
with Object, type, and member expressions	**yield** Loops, jumps, returns, and iteration		

Important classification notes

- **Reserved versus contextual:** Reserved keywords are always syntactically reserved unless escaped with @. Contextual keywords are ordinary identifiers outside the contexts where the grammar assigns them a special meaning.
- **Multiple roles:** A word can have one lexical class but several syntactic roles. For example, using can be an import directive, a using statement, or a using declaration; new can create an object, hide a member, or form a generic constructor constraint.
- **Preview status:** closed, safe, and union belong to the C# 15 preview surface as of July 30, 2026. Preview syntax and behavior can change before the final release.
- **Calling conventions:** Cdecl, Fastcall, Stdcall, and Thiscall are case-sensitive contextual words recognized in unmanaged function-pointer calling-convention lists. They are included for parser completeness even though the main Microsoft keyword index presents only managed and unmanaged.
- **Preprocessor directives:** #if, #define, #nullable, #region, and related forms are preprocessing directives, not C# keywords. Operators and punctuation such as =>, ??, ::, ., and ; are tokens, but not keywords.

Microsoft sources

- C# keywords and contextual keywords: https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/
- C# 14 (latest stable release): https://learn.microsoft.com/dotnet/csharp/whats-new/csharp-14
- C# 15 preview features: https://learn.microsoft.com/dotnet/csharp/whats-new/csharp-15
- Union types reference: https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/union
- Formal lexical grammar and calling-convention contextual words: https://learn.microsoft.com/dotnet/csharp/language-reference/language-specification/lexical-structure
- C# language versioning: https://learn.microsoft.com/dotnet/csharp/language-reference/language-versioning

Document generated July 30, 2026. Use the current Microsoft language reference when implementing a production parser, because preview keywords and grammar may change.