Glossary

.NET

The ecosystem that C# is a part of. It encompasses the .NET SDK, the compiler, the Common Language Runtime, Common Intermediate Language, the Base Class Library, and app models for building specific types of applications. (Levels 1 and 50.)

.NET Core

The original name for the current cutting-edge .NET implementation. After .NET Core 3.1, this became simply .NET. (Level 50.)

.NET Framework

The original implementation of .NET that worked only on Windows. This flavor of .NET is still used, but most new development happens on the more modern .NET implementation. (Level 50.)

.NET Multi-platform App UI

The evolution of Xamarin Forms and an upcoming cross-platform UI framework for mobile and desktop apps.

0-based Indexing

A scheme where indexes for an array or other collection type start with item number 0 instead of 1. C# uses this for almost everything.

Abstract Class

A class that you cannot create instances of; you can only create instances of classes derived from it. Only abstract classes can contain abstract members. (Level 26.)

Abstract Method

A method declaration that does not provide an implementation or body. Abstract methods can only be defined in abstract classes. Derived classes that are not abstract must provide an implementation of the method. (Level 26.)

Abstraction

The object-oriented concept where if a class keeps its inner workings private, those internal workings won't matter to the outside world. It also allows those inner workings to change without affecting the rest of the program. (Level 19.)

Accessibility Level

Types and their members indicate how broadly accessible or visible they are. The compiler will ensure that other code uses it in a compliant manner. Making something more hidden gives you more flexibility to change it later without significantly affecting the rest of the program. Making something less hidden allows it to be used in more places. The **private** accessibility level means something can only be used within the type it is defined in. The **`public`** accessibility level means it can be used anywhere and is intended for general reuse. The **`protected`** accessibility level indicates that something can only be used in the class it is defined in or derived classes. The **`internal`** accessibility level indicates that it can be used in the assembly it is defined in, but not another. The **`private protected`** accessibility level indicates that it can only be used in derived classes in the same assembly. The **`protected internal`** accessibility level can be used in derived classes or the assembly it is defined in. (Levels 19, 25, and 47.)

Accessibility Modifier

See *accessibility level*.

Ahead-of-Time Compilation

C# code is compiled to CIL instructions by the C# compiler and then turned into hardware-ready binary instructions as the program runs with the JIT compiler. Ahead-of-time compilation moves the JIT compiler's work to the same time as the main C# compiler. This makes the code operating

system- and hardware architecture-specific but speeds up initialization.

Anonymous Type

A class without a formal type name, created with the **`new`** keyword and a list of properties. E.g., **`new { A = 1, B = 2 }`**. They are immutable. (Level 20.)

AOT Compilation

See *ahead-of-time compilation*.

App Model

One of several frameworks that are a part of .NET, intended to make the development of a specific type of application (web, desktop, mobile, etc.) easy. (Level 50.)

Architecture

This word has many meanings in software development. For hardware architecture, see *Instruction Set Architecture*. For software architecture, see *object-oriented design*.

Argument

The value supplied to a method for one of its parameters.

Arm

A single branch of a switch. (Level 10.)

Array

A collection of multiple values of the same type placed together in a list-like structure. (Level 12.)

ASP.NET

An app model for building web-based applications. (Level 50.)

Assembler

A simple program that translates assembly instructions into binary instructions. (Level 49.)

Assembly

Represents a single block of redistributable code used for deployment, security, and versioning. A *.dll* or *.exe* file. Each project is compiled into its own assembly. See also *project* and *solution*. (Level 3.)

Assembly Language

A low-level programming language where each instruction corresponds directly to a binary instruction the computer can run. Essentially, a human-readable form of binary. (Level 49.)

Assignment

The process of placing a value in a variable. (Level 5.)

Associative Array

See *dictionary*.

Associativity

See *operator associativity*.

Asynchronous Programming

Allowing work to be scheduled for later after some other task finishes to prevent threads from getting stuck waiting. (Level 44.)

Attribute

A feature for attaching metadata to code elements, which can then be used by the compiler and other code analysis tools. (Level 47.)

Auto-Property

A type of property where the compiler automatically generates the backing field and basic get and set logic. (Level 20.)

Automatic Memory Management

See *managed memory*.

Awaitable

Any type that can be used with the **`await`** keyword. **`Task`** and **`Task<T>`** are the most common. (Level 44.)

Backing Field

A field that a property uses as a part of its getter and setter. (Level 20.)

Base Class

In inheritance, the class that another is built upon. The derived class inherits all members except constructors from the base class. Also called a superclass or a parent class. See also *inheritance*, *derived class*, and *sealed class*. (Level 25.)

Base Class Library

The standard library available to all programs made in C# and other .NET languages. (Level 50.)

BCL

See *Base Class Library*.

Binary

Composed of two things. Binary numbers use only 0's and 1's. (Level 3.)

Binary Code

The executable instructions that computers work with to do things. All programs are built out of binary code. (Levels 3 and 49.)

Binary Instructions

See *binary code*.

Binary Literal

A literal that specifies an integer in binary and is preceded by the marker **`0b`**: **`0b00101001`**. (Level 6.)

Binary Operator

An operator that works on two operands. Addition and subtraction are two examples. (Level 7.)

Bit

A single binary digit. A 0 or a 1. (Level 6.)

Bit Field

Compactly storing multiple related Boolean values, using only one bit per Boolean value. (Level 47.)

Bit Manipulation

Using specific operators to work with the individual bits of a data element. (Level 47.)

Bitwise Operator

One of several operators used for bit manipulation, including bitwise logical operators and bitshift operators. (Level 47.)

Block

A section of code demarcated with curly braces, typically containing many statements in sequence. (Level 9.)

Block Body

One of two styles of defining the body of a method or other member that uses a block. See also *expression body*. (Level 13.)

Boolean

Pertaining to truth values. A Boolean value can be either true or false. Used heavily in decision making and looping, and represented with the **`bool`** type in C#. (Level 6.)

Boxing

When a value type is removed from its regular place and placed elsewhere on the heap, accessible through a reference. (Level 28.)

Breakpoint

The marking of a location in code where the debugger should suspend execution so that you can inspect its state. (Bonus Level C.)

Built-In Type

One of a handful of types that the C# compiler knows a lot about and provides shortcuts to make working with them easy. These types have their own keywords, such as **`int`**, **`string`**, or **`bool`**. (Level 6.)

Byte

A block of eight bits. (Level 6.)

C++

A powerful all-purpose programming language. C++'s syntax inspired C#'s syntax. (Level 1.)

Call

See *method call*.

Callback

A method or chunk of code that is scheduled to happen after some other task completes. (Level 44.)

Casting

See t*ypecasting*.

Catch Block

A chunk of code intended to resolve an error produced by another part of the code. (Level 35.)

Character

A single letter or symbol. Represented by the **`char`** type. (Level 6.)

Checked Context

A section of code wherein mathematical overflow will throw an exception instead of wrapping around. An unchecked context is the default. (Level 47.)

CIL

See *Common Intermediate Language*.

Class

A category of types, formed by combining fields (data) and methods (operations on that data). The most versatile type you can define. Creates a blueprint used by instances of the type. All classes are reference types. See also *struct*, *type*, *record*, and *object*. (Level 18.)

Closure

The ability for certain functions (lambdas and local functions) to have access to variables defined in the context around them without having to pass them in as arguments. (Level 38.)

CLR

See *Common Language Runtime*.

Code Window

The main window in Visual Studio. Allows you to edit code. (Bonus Level A.)

Collection Initializer Syntax

A way to declare and populate a collection type by listing the contents between curly braces: `new int[] { 1, 2, 3 }`.

Command Line Arguments

Arguments passed to a program as it launches. (Level 47.)

Comment

Annotations placed within source code, intended to be read by programmers but ignored by the compiler. (Level 4.)

Common Intermediate Language

The compiled language that the Common Language Runtime processes and the target of the C# compiler. A high-level, object-oriented form of assembly code. (Level 50.)

Common Language Runtime

The runtime and virtual machine that C# programs run on top of. (Level 50.)

Compilation Unit

See *assembly*.

Compile-Time Constant

A value that the compiler can compute ahead of time. A literal `0` and the expression `2 + 3` are both compile-time constants. See also *constant*.

Compiler

A program that turns source code into executable machine code. (Levels 3 and 49.)

Compiler Error

An indication from the compiler that it cannot translate your C# code into instructions that the computer can run. (Bonus Level B.)

Compiler Warning

An indication from the compiler that it suspects a mistake has been made, even though it could technically produce binary instructions from your code. (Bonus Level B.)

Composite Type

A type made by assembling other elements to form a new type. Tuples, classes, and structs are all composite types. (Level 17.)

Compound Assignment Operator

An operator that combines another operation with assignment, such as `x += 3;` (Level 7.)

Concrete Class

A class that is not abstract. (Level 26.)

Concurrency

Using threads to run multiple things at the same time. (Level 43.)

Conditional Compilation Symbol

A flag that the compiler can use to decide whether to include a section of code. (Level 47.)

Constant

A variable-like construct whose value is computed by the compiler and cannot change as the program runs. Consider also a `readonly` field, which is negligibly slower but substantially more flexible. (Level 47.)

Constructor

A special category of methods designed to initialize new objects into a valid starting state. (Level 18.)

Context Switch

In multi-threaded programming, the overhead needed to save the state of an active thread and replace it with previously saved state from another thread. (Level 43.)

Contravariance

See *generic variance*.

Covariance

See *generic variance*.

CRC Card

A technique for doing object-oriented design using pen and paper to allow for rapid decision making and brainstorming before writing code. (Level 23.)

Critical Section

A block of code that should not be accessed by more than one thread at once. Critical sections are usually blocked off with a mutex to prevent simultaneous thread access. (Level 43.)

Curly Braces

The symbols **{** and **}**, used to mark blocks of code. (Level 3.)

Custom Conversion

Defining a conversion from one type to another. (Level 41.)

Dangling Pointer

A memory error where the memory is no longer in use, but a part of the program attempts to use it still. Solved in C# via managed memory and garbage collection. (Level 14.)

Data Structure

A long name to refer to a struct. (Level 28.) Alternatively, a word for any type that is primarily about storing data, whether it is a class or a struct.

Deadlock

When a thread is perpetually waiting to acquire a lock but will never be able to acquire it because another thread has acquired it and is stalled. (Level 43.)

Debug

The process of working through your code to find and fix problems. (Bonus Level C.)

Debugger

A tool aimed at facilitating debugging, which attaches itself to a running program and allows you to suspend the program and inspect its state. (Bonus Level C.)

Declaration

The definition of a variable, method, or other code element. (Level 5.)

Deconstruction

Extracting the elements from a tuple or other type into separate individual variables. (Level 17.)

Decrementing

Subtracting 1 from a variable. See also *incrementing*. (Level 7.)

Deferred Execution

Code that defines what needs to be computed but runs as little as necessary to produce the next result. Usually said of query expressions. (Level 42.)

Delegate

A variable that contains a reference to a method. Treats code as data. (Level 36.)

Dependency

A separate library (frequently a *.dll*) that another project references and utilizes. The project is said to "depend on" the referenced library. (Level 48.)

Derived Class

In inheritance, the class that builds upon another. The derived class inherits all members except constructors from the base class. Also called a subclass or a child class. See also *inheritance* and *base class*. (Level 25.)

Deserialization

See *serialization*.

Design

See *object-oriented design*.

Dictionary

A data structure that allows items to be found by some key. (Level 32.)

Digit Separator

An underscore character (**_**) placed between the digits of a numeric literal, used to organize the digits more cleanly, without changing the number's meaning. E.g., 1_543_220. (Level 6.)

Discard

An underscore character (**_**), used in places where a variable is expected, that tells the compiler to generate a throwaway variable in its place. Also used in pattern matching to indicate that anything is a match. (Levels 34 and 40.)

Divide and Conquer

The process of taking a large, complicated task and breaking it down into more manageable, smaller pieces. (Level 3.)

Division by Zero

An attempt to use a value of 0 on the bottom of a division operation. Mathematically illogical, attempting to do this in C# code can result in your program crashing. (Level 7.)

DLL

A specific type of assembly without a defined entry point. Intended to be reused by other applications. See also *assembly* and *EXE*. (Level 48.)

Dynamic Object

An object whose members are not known until run-time and may even be changeable at run-time. Should be stored in a variable of type **`dynamic`** to allow dynamic type checking. (Level 45.)

Dynamic Type Checking

Checking if members such as methods and properties exist as the program is running, instead of at compile time. See also *static type checking*. (Level 45.)

E Notation

A way of expressing very large or tiny numbers in a modified version of scientific notation (e.g., 1.3×10^{31}) by substituting the multiplication and 10 base with the letter 'E' (e.g., "1.3e31"). **`float`**, **`double`**, and **`decimal`** can all be expressed with E notation. (Level 6.)

Early Exit

Returning from a method before the last statement executes. (Level 13.)

Encapsulation

Combining fields (data) and methods (operations on the data) into a single cohesive bundle. A fundamental principle of object-oriented programming. See also *class*. (Level 18.)

Entry Point

The place in the code where the program begins running. The main method. (Level 3.)

Enum

See *enumeration*.

Enumeration

A type definition that names off each allowed choice. (Level 16.)

Error List

A window in Visual Studio that displays a list of compiler errors and warnings. (Bonus Level A.)

Evaluation

To compute what an expression represents. The expression **`2 * 3 - 1`** evaluates to **`5`**. (Level 3.)

Event

A mechanism that allows one object to notify others that something has happened. (Level 37.)

Event Leak

When an object unintentionally remains in memory solely because one of its methods is still attached to an event. (Level 37.)

Exception

An object that encapsulates an error that occurred while executing code. The object is "thrown" or passed up the call stack to the calling method until it is either handled ("caught") or reaches the top of the call stack, causing the program to crash. (Level 35.)

EXE

A specific type of assembly containing an entry point. See also *assembly* and *DLL*. (Level 48.)

Explicit

A term used in programming to mean that something is formally stated or written out. The opposite of *implicit*.

Expression

A chunk of code that your program evaluates to compute a single value. A fundamental building block of C# programming. (Level 3.)

Expression Body

One of two styles of defining the body of a method, constructor, property, etc., that uses a single expression. See also *block body*. (Level 13.)

Extension Method

A type of static method that can be called as though it is an instance method of another type. (Level 34.)

Field

A variable declared as a member of a class or other type, as opposed to a local variable or parameter. For fields not marked static, each instance will have its own copy. Making fields **`private`** facilitates the principle of abstraction. Sometimes called instance variables. (Level 18.)

Fixed-Size Array

An array whose size is always the same and whose memory is allocated as a part of the struct it lives within instead of elsewhere on the heap. Used primarily for interoperating with unmanaged code. (Level 46.)

Fixed Statement

A statement that causes a reference to be "pinned" in place, temporarily barring the garbage collector from moving it. Only allowed in unsafe contexts. (Level 46.)

Floating-Point Division

The type of division used with floating-point types. It can produce fractional values. With floating-point division, `3.0/2.0` equals `1.5`. Contrasted with *integer division*. (Level 7.)

Floating-Point Type

One of several built-in types used for storing real-valued (non-integer) numbers like 2.36. `float`, `double`, and `decimal` are all floating-point types. (Level 6.)

`for` Loop

See *loop*.

`foreach` Loop

See *loop*.

Frame

See s*tack frame*.

Framework-Dependent Deployment

A deployment where only your code is included, without the .NET runtime. Assumes the target .NET version is already installed on the destination machine. (Level 51.)

Fully Qualified Name

The full name of a type, including the namespace it belongs in. (Level 33.)

Function

A chunk of reusable code that does some specific task. The terms *method* and *function* are often treated as synonyms, but more specifically, a method is a function declared as a member of a class or other type. Other types of functions include local functions and lambdas. (Level 13.)

Garbage Collection

The process of removing objects on the heap that are no longer accessible. Garbage collection in C# is automatic. See also *managed memory*. (Level 14.)

Generic Type Argument

A specific type used to fill in a generic type parameter. For the generic type `List<T>`, when creating a `new List<int>()`, `int` is the generic type argument being used for the generic type parameter `T`. (Level 30.)

Generic Type Constraint

A rule limiting which types can be used as a generic type argument for some specific generic type parameter. For example, a constraint can require that the type argument implement a particular interface, have a parameterless constructor, be a reference type, etc. (Level 30.)

Generic Type Parameter

In generics, a placeholder for a type to be filled in later. `T` is a generic type parameter in the type `List<T>`. (Level 30.)

Generic Type

A type that leaves a placeholder for certain types used within it. The placeholders are called generic type parameters. `List<T>` and `Dictionary<TKey, TValue>` are both examples of generic types. (Level 30.)

Generic Variance

A mechanism for specifying hierarchy-like relationships among generic types. Even if `Derived` is derived from `Base`, `Generic<Derived>` is not derived from `Generic<Base>`. Generic variance determines when one type can be used in place of the other. Invariance is the default and indicates that neither can be used in place of the other. Covariance allows the generic types to mirror the type parameter's inheritance, allowing `Generic<Derived>` to be used in places where `Generic<Base>` is expected. Contravariance allows the generic types to invert the type parameter's inheritance, allowing `Generic<Base>` where `Generic<Derived>` is expected. (Level 47.)

Getter

A method—especially the `get` part of a property—that returns a value that represents a part of an object's state. (Levels 19 and 20.)

Global Namespace

The root namespace. Where type definitions live if not placed in a specific namespace. (Level 33).

Global State

A variable that can be accessed from anywhere in the program. For example, a `public static` field. Usually considered bad practice because otherwise independent parts of your program become closely intertwined through the global state. (Level 21.)

Hash Code

A number generated for an object, used for fast lookups in types like dictionaries. Overridable through the `GetHashCode` method. (Level 32.)

Heap

A section of memory where new data can be placed as the need arises. One of two main parts of a program's memory. Memory allocation and deallocation must be handled carefully, which C# does through managed memory and the garbage collector. See also *stack* and *reference type*. (Level 14.)

Hexadecimal

A base-16 numbering scheme, typically representing a single digit with the symbols 0 through 9 and A through F.

Popular in computing because it can compactly represent a byte's contents with only two characters. (Level 6.)

Hexadecimal Literal

A numeric literal written in hexadecimal, preceded by a **`0x`**: **`0xac915d6c`**. (Level 6.)

IDE

See i*ntegrated development environment*.

IL

See *Common Intermediate Language*.

Immutability

Not able to be changed once constructed. Said of a field or type. (Level 20.)

Implicit

A term frequently used to mean something happens without needing to be expressly stated. The opposite of *explicit*.

Incrementing

Adding 1 to a variable. See also *decrementing*. (Level 7.)

Index

A number used to refer to a specific item in an array or other collection type. (Level 12.)

Indexer

A type member that defines how the type should treat indexing operations. (Level 41.)

Inference

See *type inference*.

Infinite Loop

A loop that never ends. Usually considered a bug, but some situations intentionally take advantage of a neverending loop. **`while (true) { ... }`**. (Level 11.)

Inheritance

The ability for one class to build upon or derive from another. The new derived class keeps (inherits) all members from the original base class and can add more members. (Level 25.)

Initialization

Assigning a starting value to a variable. Local variables cannot be used until they have a value assigned to them. Parameters are initialized by the calling method, and fields are initialized to a bit pattern of all 0's when the object is first constructed. (Level 5.)

Instance

An object of a specific type. A **`string`** instance is an object whose type is **`string`**. See also *object*. (Level 15.)

Instance Variable

See *field*.

Instruction Set Architecture

A standardized set of instructions that a computer's CPU can run. x86/x64 and ARM are the two most popular. (Level 49.)

Integer Division

A style of division, used with C#'s integer types, where fractional values are discarded. **`3/2`** is **`1`**, with the additional **`0.5`** being discarded. (Level 7.)

Integral Type

A type that represents an integer: **`byte`**, **`short`**, **`int`**, **`long`**, **`sbyte`**, **`ushort`**, **`uint`**, and **`ulong`**. It also includes the **`char`** type. (Level 6.)

Integrated Development Environment

A program designed for making programming easy. They assemble all of the tools needed together into a single cohesive program. Visual Studio is an example. (Levels 2 and A.)

IntelliSense

A Visual Studio feature that instantly performs services for you as you type, including name completion and displaying documentation about what you are typing. (Bonus Level A.)

Interface

A type that defines a set of capabilities or responsibilities that another type must have. Sometimes also used to refer to the public parts of any type (signatures, but not implementations). (Level 27.)

Internal

See *accessibility level*.

Invariance

See *generic variance*.

Invoke

See *method call*.

ISA

See *Instruction Set Architecture*.

Iterator Method

A method that uses `yield return` to produce an `IEnumerable<T>` a little at a time as needed. (Level 47.)

Jagged Array

An array of arrays. Each array within the main array can be a different length. (Level 12.)

Java

A high-level, all-purpose programming language similar to C#. Like C#, it also runs on a virtual machine. (Level 3.)

JIT Compiler

See *Just-in-Time Compiler*.

Just-in-Time Compiler

A compiler that translates CIL instructions to binary instructions that the computer can execute directly. This typically happens as the program runs, method-by-method as the method is first used, leading to the name "just in time." (Level 50.)

Keyword

A word that has specific meaning within a programming language. (Level 3.)

Lambda Expression

An anonymous single-use method, written with simplified syntax to make it easy to create. Often used when delegates are needed. (Level 38.)

Language-Integrated Query

A part of the C# language that allows you to perform queries on collections within your program. These queries involve taking a data set and then filtering, combining, transforming, grouping, and ordering it to produce the desired result set. Often called LINQ (pronounced "link"). (Level 42.)

Lazy Evaluation

When the program runs only enough of an expression to determine an answer. Primarily used with logical expressions. The expression `a && b` will not evaluate `b` if `a` is `false` because the overall answer is already known. (Level 9.)

Left Associativity

See *operator associativity*.

LINQ

See *Language Integrated Query*.

Literal

A fixed, directly stated value in source code. For example, in the line `int x = 3;`, the `3` is an `int` literal. You can create literals of all of the built-in types without needing to use `new`. (Level 5.)

Local Function

A function that is contained directly in another function or method. These are only accessible within the method they are defined in. (Level 34.)

Local Variable

A variable created inside a method and only accessible within that method. Some local variables are declared inside a block, such as a loop, and have a narrower scope than the entire method. (Level 5.)

Logical Operator

The operators `&&`, `||`, and `!` (*and*, *or*, and *not*, respectively) used in logic expressions. (Level 9.)

Loop

To repeat something multiple times. C# has various loops, including the `for`, `while`, `do`/`while`, and `foreach` loops. (Level 11.)

Main Method

The entry point of an application. The code that runs when the program is started. (Level 3.)

Managed Code

Code whose memory allocation and cleanup is managed by its runtime. Most C# code is managed code. See also *managed memory* and *unmanaged code*. (Level 14.)

Managed Memory

Memory whose allocation and cleanup are managed automatically by the runtime through garbage collection. In a C# program, the heap is considered managed memory. (Level 14.)

MAUI

See *.NET Multi-platform App UI*.

Map

See *dictionary*.

Member

Broadly, anything that is defined inside of something else. Usually refers to members of a type definition, such as fields, methods, properties, and events. (Level 3.)

Memory Address

A number that represents a specific location in memory. (Level 5.)

Memory Allocation

Reserving a location in memory to hold specific data for a newly created object or value. (Level 14.)

Memory Leak

Occurs when a program fails to clean up memory it is no longer using. Memory leaks often lead to consuming more and more memory until none is left. C# uses a managed memory scheme, which largely avoids this problem. (Level 14.)

Memory Safety

The ability of .NET's managed memory system to ensure that all accesses to memory contain legitimate living objects and data. (Level 50.)

Method

A section of code that accomplishes a single job or task in the broader system. Methods have a name, a list of parameters for supplying data to the method, and a return value to track its result. (Level 13.)

Method Body

See *method implementation.*

Method Call

The act of pausing execution in one method, jumping over to a second method, and running it to completion before returning to the original method to resume execution. Method calls are tracked on the stack. (Level 13.)

Method Call Syntax

A way of performing LINQ queries using method calls instead of the LINQ keywords. Contrast with *query syntax.* (Level 42.)

Method Group

The collection of all methods within a type that share the same name. All overloads of a method. (Level 13.)

Method Implementation

The code that defines what a method should do when it is called. (Level 13.)

Method Invocation

See *method call.*

Method Overload

Defining two or more methods with the same name but differing in number or types of parameters. Overloads should perform the same conceptual task, just with different arguments. (Level 13.)

Method Signature

A method's name and the number and types of its parameters. This does not include its return type, nor does it include parameter names. This is primarily how methods are distinguished from one another. Two methods in a type cannot share the same method signature. (Level 13.)

Mono

An open source .NET implementation, but that runs on non-Windows platforms. Largely superseded by .NET Core and .NET. (Level 50.)

Multi-Dimensional Array

An array that contains items in a 2D grid (or three or more dimensions) instead of just a single row. (Level 12.)

Multi-Threading

Using more than one thread of execution to run code simultaneously. (Level 43.)

Mutex

See *mutual exclusion.*

Mutual Exclusion

Structuring code so that only one thread can access it at a time. The mechanism that forces this is often called a mutex. (Level 43.)

Name Collision

Occurs when two types share a name and need to be disambiguated with fully qualified names or an alias. (Level 33.)

Name Hiding

When a local variable or parameter shares the same name as a field, making the field not directly accessible. The field may still be accessed using the **`this`** keyword. (Level 18.)

Named Argument

Listing parameter names associated with an argument. This allows arguments to be supplied out of order. (Level 34.)

Namespace

A grouping of types under a shared name. (Level 33.)

NaN

A special value used by floating-point types to indicate a computation resulted in an undefined or unrepresentable value. It stands for "Not a Number." (Level 7.)

Narrowing Conversion

A conversion from one type to another that loses data in the process. Casting from a `long` to an `int` is a narrowing conversion. See also *widening conversion*. (Level 7.)

Native Code

See *Unmanaged Code*.

Nesting

Placing a code element inside another of the same type, such as nested parentheses, nested loops, nested `if` statements, and nested classes. (Level 9.)

Noun and Verb Extraction

Marking the nouns and verbs in a set of requirements, used as a way to begin the software design process. (Level 23.)

NuGet Package Manager

A tool for making it easy to find and use code created by other programmers (bundled into packages) in your program. (Level 48.)

Null Reference

A special reference that refers to no object at all. (Level 22.)

Nullable Value Type

A mechanism for allowing value types to express a lack of a value (`null`). Done by placing a `?` after a value typed-variable: `int? a = null;` (Level 32.)

Object

An element in an object-oriented system, usually given a single, focused responsibility or set of related responsibilities. In C#, all objects belong to a specific class, and all objects of the same class share the same structure and definition. (Level 18.)

Object-Initializer Syntax

The ability to set values for an object's properties immediately after a constructor runs. (Level 20.)

Object-Oriented Design

Deciding how to split a large program into multiple objects and how to have them coordinate with each other. (Level 23.) See also *object-oriented programming*.

Object-Oriented Programming

An approach to programming where the functionality is split across multiple components called objects, each responsible for a slice of the overall problem and coordinating with other objects to complete the job. C# is an object-oriented programming language. (Level 15.)

Operation

An expression that combines other elements into one, using symbols instead of names to indicate the operation. `+`, `-`, `*`, and `/` are all operations. (Level 7.)

Operator

A symbol that denotes a specific operation, such as the `+`, `==`, or `!` operators. (Level 7.)

Operator Associativity

The rules that determine the order that operations of the same precedence are evaluated in. This is either left-to-right or right-to-left. For example, in the expression `5 - 3 - 1`, `5 - 3` is evaluated first because subtraction's associativity is left-to-right. (Level 7.)

Operator Overloading

Defining what an operator should do for some specific type. (Level 41.)

Operator Precedence

The rules that determine which operations should happen first. For example, multiplication has higher precedence than addition and should be done first. (Level 7.)

Optional Parameter

A method parameter with a default value. Allows the method to be called without supplying a value for that parameter. (Level 34.)

Order of Operations

The rules determining the order that operations are applied in when an expression contains more than one, determined first by operator precedence and second by operator associativity. (Level 7.)

Out-of-Order Execution

The compiler or hardware's ability to reorder instructions for performance reasons as long as the code still behaves as though the statements happen in the order they are written in. (Level 47.)

Overflow

When the result of an operation exceeds what the data type can represent. (Level 7.)

Overload

For overloading methods, see *method overload*. For overloading operators, see *operator overloading*.

Overload Resolution

The rules the compiler uses when determining which of many candidate methods is intended by C# code. (Level 13.)

Override

When a derived class supplies an alternative implementation for a method defined in the base class. (Level 26.)

P/Invoke

See *Platform Invocation Services*.

Package

A bundle of compiled code (usually a *.dll*) and metadata that can be referenced and managed with the NuGet Package Manager. (Level 48.)

Parameter

A type of variable with method scope like a local variable, but whose initial value is supplied by the calling method. (Level 13.)

Parent Class

See *base class*.

Parentheses

The symbols `(` and `)`, used for forcing an operation to happen outside of its standard order, the conversion operator, and method calls. (Levels 7 and 13.)

Parse

Taking text and breaking it up into small pieces that have individual meaning. Parsing is often done when reading content from a file or interpreting user input. (Level 8.)

Partial Class

A class that is defined across multiple sections, usually across multiple files. (Level 47.)

Passing by Reference

Said of a method parameter, when parameters do not represent a new memory location but an alias for an existing memory location elsewhere. Typically done in C# with the **`ref`** or **`out`** keywords. Contrast with *passing by value*. (Level 34.)

Passing by Value

Said of a method parameter, when parameters represent new memory locations. The data supplied to the method is copied into the parameter's memory location. Most things in C# are passed by value, including both value types and reference types, though with reference types, a copy of the reference is made rather than a copy of the entire object. Contrast with *passing by reference*. (Level 34.)

Pattern Matching

A code element that allows for placing data into one of several categories based on the object's structure and properties. Used in switches and with the **`is`** keyword. (Level 40.)

Pinning

See *fixed statement*.

Platform Invocation Services

A mechanism that lets your C# code directly invoke unmanaged code that lives in another DLL referenced by your project. (Level 46.)

Pointer

A variable type that contains a raw memory address. Philosophically similar to a reference, but sidesteps the managed memory system. Used only in unsafe code and intended for working with native code. (Level 46.)

Polymorphism

The ability for a base class to define a method that derived classes can then override. As the program is running, it will invoke the correct version of the method as determined by the type of the object, not the type of the variable, allowing for different behavior depending on the class of the object involved. (Level 26.)

Public

See *accessibility level*.

Precedence

See *operator precedence*.

Preprocessor Directive

Special instructions for the compiler embedded in source code. (Level 47.)

Primitive Type

See *built-in type*.

Print Debugging

Using **`Console.WriteLine`** and similar calls to diagnose what your program is doing. Contrasted with using a debugger. (Bonus Level C.)

Private

See *accessibility level*.

Procedure

See *method*.

Program Order

The order that statements are written in the source code. In C#, it is assumed that these instructions will execute from top to bottom, though the compiler and hardware may make optimizations that change the order, so long as the effect is the same. See also *out-of-order execution* and *volatile field*. (Level 47.)

Project

A collection of source code, resource files, and configuration compiled together into the same assembly (DLL or EXE). See also *solution* and *assembly*. (Levels 3 and 48.)

Property

A member that provides field-like access while still allowing the class to use information hiding to protect its data from direct access. (Level 20.)

Protected

See *accessibility level*.

Query Expression

A type of expression that allows you to make queries on a collection to manipulate, filter, combine, and reorder the results. Query expressions are a fundamental part of LINQ. (Level 42.)

Query Syntax

One of the two flavors of LINQ (contrasted with method call syntax) that uses keywords and clauses to perform queries against data sets. (Level 42.)

Record

A compact way to define data-centric classes (or structs). (Level 29.)

Rectangular Array

See *multi-dimensional array*.

Recursion

A method that calls itself. To avoid running out of memory, care must be taken to ensure progress towards a base case is being made. See also *recursion*. (Level 13.)

Refactor

Changing source code in a way that doesn't change the software's external behavior to improve other qualities of the code, such as readability and maintainability. (Level 23.)

Reference

A unique identifier for an object on the heap, used to find an object located there. (Level 14.)

Reference Semantics

When two things are considered equal only if they are the same object in memory. Contrasts with *value semantics*. (Level 14.)

Reference Type

One of two main categories of types where the data for the variable lives somewhere on the heap and variables contain a reference used to retrieve the data on the heap. Classes are all reference types, as are strings, objects, and arrays. See also *value type* and *reference*. (Level 14.)

Reflection

The ability of a program to inspect code (types and their members) as the program runs. (Level 47.)

Relational Operators

Operators that determine a relationship between two values, such as equality (`==`), inequality (`!=`), or less than or greater than relationships. (Level 9.)

Requirements Gathering

The process of determining what should be built. (Level 23.)

Return

The process of going from one method back to the one that called it. Also used to refer to providing a result (a return value) as a part of returning. (Level 13.)

Return Type

The data type of the value returned by a method or `void` to indicate no return value. (Level 13.)

Right Associativity

See *operator associativity*.

Roundoff Error

When an operation results in loss of information with floating-point types because the value was too small to be represented. (Level 7.)

Run-Time Constant

See *constant*.

Runtime

Code provided by the language and compiler that performs the job that the programming language promised to do. C#'s runtime is called the Common Language Runtime. (Level 49.) Also used to describe something that happens as the program is running.

Scheduler

A component of the operating system that decides when threads should run. (Level 43.)

Scientific Notation

The representation of very large or small numbers by expressing them as a number between 1 and 10, multiplied by a power of ten. E.g., "1.3×10^{31}." In C# code, this is usually expressed through E Notation. (Level 6.)

Scope

The part of the code in which a named code element (variable, method, class, etc.) can referred to. (Level 9.)

Sealed Class

A class that prohibits other classes from using it as a base class. (Level 25.)

Self-Contained Deployment

A packaged version of the software which includes a copy of the runtime of the target machine so that the runtime does not need to be installed separately. (Level 51.)

Signed Type

A numeric type that includes a **+** or **−** sign. (Level 6.)

Software Design

See *object-oriented design*.

Solution

A collection of one or more related projects that work together to form a complete product. See also *project* and *Solution Explorer*. (Levels 3 and 48.)

Solution Explorer

A window in Visual Studio that outlines the overall structure of the code you are working on. (Bonus Level A.)

Source Code

Human-readable instructions for the computer that will ultimately be turned into something the computer can execute. (Levels 1 and 49.)

Square Brackets

The symbols **[** and **]**, used for array indexing. (Level 12.)

Stack

A section of memory where data is allocated based on method calls and their local variables and parameters. Memory is allocated by adding a frame when a method is called and cleaned up automatically when it returns. See also *heap* and *value type*. (Level 14.)

Stack Allocation

The placement of an array's memory on the stack instead of the heap. This can only be done in an unsafe context and only with local array variables. (Level 46.)

Stack Frame

A section of memory on the stack that holds the local variables and parameters of a single method, along with metadata that allows it to remember where to return to. (Level 14.)

Stack Trace

A representation of all of the frames on the stack that indicates the current state of execution within a program. (Level 35.)

Standard Library

The collection of code that is available for all programs written in a particular language. C#'s standard library is also called the Base Class Library. (Level 50.)

Statement

A single step in a program. C# programs are formed from many statements placed one after the next. (Level 3.)

Static

Not belonging to a specific instance, but the type as a whole. Many member types can be static, including methods, fields, properties, and constructors. (Level 21.)

Static Type Checking

When the compiler checks that objects placed into variables match the types and ensures the existence of the methods, properties, and operators used in code. (Level 45.)

String

Text. A sequence of characters. Represented by the **`string`** type. (Level 6.)

Struct

A custom-made value type, with many (but not all) of the same features as a class. (Level 28.)

Subclass

See *derived class*.

Superclass

See *base class*.

Subroutine

See *method*.

Switch

A language construct where one branch of many is chosen based on conditions supplied for each branch. (Level 10.)

Synchronous Programming

Code where each operation runs to completion before moving to the next. The usual type of code. Contrast with *asynchronous programming*. (Level 44.)

Task

One of several types that represents an asynchronous chunk of work that happens behind the scenes. C# has powerful language features that let you construct asynchronous code out of tasks without making the code much harder to understand. (Level 44.)

Ternary Operator

An operator that works on three operands. C# only has one ternary operator, which is the conditional operator. (Level 9.)

Thread

A component of a process that can execute instructions. All programs use at least one thread, while some use many. (Level 43.)

Thread Pool

A collection of threads, automatically managed by the runtime, which runs most tasks. (Level 44.)

Thread Safety

Ensuring that parts of code that should not be accessed simultaneously by multiple threads (critical sections) are not accessible by multiple threads. (Level 43.)

Top-Level Statement

Statements that exist outside of other classes and methods that form the main method. (Levels 3 and 33.)

Tuple

A simple data structure that stores a set of related values of different types as a single unit. (Level 17.)

Type

A category of values or objects that defines what data it can represent and what you can do with it. Defining classes, structs, records, enumerations, interfaces, and delegates all define new types. (Levels 6 and 14.)

Type Inference

The C# compiler's ability to figure out the type being used in certain situations, allowing you to leave off the type (or use the **var** type). (Level 6.)

Type Safety

The compiler and runtime's ability to ensure that there is no way for one type to be mistaken for another. This plays a critical role in generics. (Level 6.)

Typecasting

Converting from one type to another using the conversion operator: `int x = (int)3.4;` (Level 7.)

Unary Operator

An operator that works with only a single value, such as the negation operator (the **–** sign in the value **–3**). (Level 7.)

Unboxing

See *boxing*.

Unchecked Context

A section of code wherein mathematical overflow will wrap around instead of throwing an exception. An unchecked context is the default. (Level 47.)

Underlying Type

The type that an enumeration is based on. (Level 16.)

Universal Windows Platform

The newest native desktop app model. Abbreviated UWP. (Level 50.)

Unmanaged Code

Code that does not have automatic memory management, and where the programmer must know when to allocate and deallocate memory. C# can work with code written in an unmanaged language (such as C or C++) but must use unsafe code. Contrast with *managed code*. (Level 46.)

Unpacking

See *deconstruction*.

Unsafe Code

Stepping outside the bounds of how the runtime manages memory for you, giving you tools for direct management and manipulation of memory. Primarily used for working with unmanaged code. (Level 46.)

Unsafe Context

A region of code wherein unsafe code can be used. (Level 46.)

Unsigned Type

A type that does not include a **+** or **–** sign (generally assumed to be positive). (Level 6.)

Unverifiable Code

See *unsafe code*.

User-Defined Conversion

A type conversion defined for types that you create. See also *typecasting*. (Level 41.)

`using` Directive

A special statement at the beginning of a C# source file that identifies namespaces the compiler should look in for simple type names used throughout the file. (Level 33.)

`using` Statement

A statement that cleanly disposes of objects that implement the **`IDisposable`** interface. (Level 47.)

Value Semantics

When two things are considered equal if their data members are all equal. Records, structs, and other value types have value semantics by default. Classes can override equality methods and operators to give them value semantics. Contrasts with *reference semantics*. (Level 14.)

Value Type

One of two main categories of types. A value-typed variable will contain its data directly in that location without any references. Structs, enumerations, **`bool`**, **`char`**, and all numeric types are value types. Contrast with *reference type*. (Level 14.)

Variable

A named memory location with a known type. Once created, a variable's name and type cannot change, though its contents can (unless intentionally made read-only). Local variables, parameters, and fields are all types of variables. (Level 5.)

Variance

See *generic variance*.

Virtual Machine

A software program that runs a virtual instruction set, typically by compiling the virtual instructions to actual hardware instructions as it is running. In the .NET world, the Common Language Runtime is a virtual machine whose instruction set is the Common Intermediate Language. C# programs are compiled from C# source code into CIL code, which the runtime executes. See also *just-in-time compiler*, *Common Intermediate Language*, and *Common Language Runtime*. (Level 49.)

Virtual Method

A method that can be overridden in derived classes. The version used is determined as the program runs instead of by the compiler to take advantage of polymorphism. Virtual methods are marked with the **`virtual`** keyword. See also *abstract method* and *overriding*. (Level 26.)

Visual Basic

Another popular language in the .NET ecosystem. While it shares many of the same capabilities as C#, it is very different in syntax. (Level 1.)

Visual Studio

Microsoft's IDE, designed for making programs in C# and other programming languages. (Level 2 and Bonus Level A.)

Visual Studio Code

A lightweight cross-platform code editor that can be used to make C# programs. (Level 2.)

Volatile Field

A field marked with the **`volatile`** keyword, which will prevent any out-of-order execution optimizations that affect behavior in a multi-threaded application. (Level 47.)

Windows Forms

The oldest app model for desktop development. Often called WinForms. (Level 50.)

Windows Presentation Foundation

An app model for desktop development. Often abbreviated WPF. (Level 50.)

Xamarin Forms

An app model for cross-platform apps. Historically, the focus was mobile app development, but future incarnations with the new name .NET Multi-platform App UI or .NET MAUI will also work for desktop applications. (Level 50.)

XML Documentation Comment

A type of comment placed above type definitions and type member definitions. These comments have a specific structure, which allows tools like Visual Studio to interpret the comments and provide automatic documentation about those types and type members. (Level 13.)