C#/.NET Learning Notes (Compiled)

C# Basics: Data Types (Primitive, Value vs Reference)

C# Basics: Data Types (Primitive, Value vs Reference)

What are Data Types?

Data types define the kind of data a variable can hold in a programming language. In C#, data types are crucial because they determine how much memory is allocated and what operations can be performed on the data.

Categories of Data Types in C#

1. Primitive (Built-in) Types: These are basic types provided by the language, such as int, double, char, and bool.
2. Value Types: These types store data directly. Examples include all primitive types (except string), structs, and enums. Value types are usually stored on the stack.
3. Reference Types: These types store a reference (address) to the actual data. Examples include string, arrays, classes, and delegates. Reference types are stored on the heap, and variables hold a reference to the memory location.

Value vs Reference Types

- Value Types: When you assign a value type variable to another, a copy of the value is made. Changes to one variable do not affect the other.
- Reference Types: When you assign a reference type variable to another, both variables refer to the same object in memory. Changes to one variable affect the other.

Why is this important?

Understanding the difference helps you predict how your data will behave when passed to methods or assigned to new variables, which is essential for writing bug-free code.

Further Reading

- Microsoft Docs: Types in C#: https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/built-in-types (https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/built-in-types)
- Microsoft Docs: Value Types and Reference Types: https://learn.microsoft.com/dotnet/csharp/programming-guide/types/ (https://learn.microsoft.com/dotnet/csharp/programming-guide/types/)

C# Basics: Variables, Operators, and Expressions

C# Basics: Variables, Operators, and Expressions

Variables

Variables are named storage locations in memory that hold data. In C#, you must declare a variable with a specific data type before using it. This helps the compiler allocate the right amount of memory and enforce type safety.

Key Points:

- Variables must be declared before use.
- The data type determines what kind of data the variable can store.
- Variable names should be descriptive and follow C# naming conventions (camelCase for local variables).

Operators

Operators are symbols that perform operations on variables and values. C# includes several types of operators:

- Arithmetic Operators: For mathematical operations (e.g., +, -, *, /, %)
- Assignment Operators: For assigning values (e.g., =, +=, -=)
- Comparison Operators: For comparing values (e.g., ==, !=, <, >, <=, >=)
- Logical Operators: For logical operations (e.g., &&, ||, !)

Expressions

An expression is a combination of variables, values, and operators that produces a result. For example, a + b is an expression that adds two variables.

Best Practices

- Use meaningful variable names.
- Keep expressions simple and readable.
- Use parentheses to clarify complex expressions.

Further Reading

- Microsoft Docs: Variables: https://learn.microsoft.com/dotnet/csharp/programming-guide/variables/ (https://learn.microsoft.com/dotnet/csharp/programming-guide/variables/)
- Microsoft Docs: Operators: https://learn.microsoft.com/dotnet/csharp/language-reference/operators/ (https://learn.microsoft.com/dotnet/csharp/language-reference/operators/)
- Microsoft Docs: Expressions: https://learn.microsoft.com/dotnet/csharp/language-reference/operators/expressions (https://learn.microsoft.com/dotnet/csharp/language-reference/operators/expressions)

Type Conversion in C# (Implicit/Explicit, Boxing/Unboxing)

Type Conversion in C# (Implicit/Explicit, Boxing/Unboxing)

Why Conversion Matters

C# is statically typed, so types must match. Conversions let values move between compatible types with predictable behavior.

Implicit vs Explicit Conversion

- Implicit conversions are safe and lossless (e.g., smaller numeric type to larger). The compiler applies them automatically.
- Explicit conversions require intent because information may be lost or the conversion may fail at runtime.

Numeric Conversions

- Widening (safe): smaller range/precision to larger range/precision.
- Narrowing (risky): larger to smaller; may overflow, truncate, or throw at runtime if checked.

Reference Conversions

- Upcast (derived to base) is safe conceptually.
- Downcast (base to derived) requires runtime type compatibility.

Boxing/Unboxing

- Boxing: wrapping a value type instance as an object to treat it as a reference type.
- Unboxing: extracting the value type from an object; requires the exact original value type.
- Performance note: boxing allocates on the heap and can pressure GC; avoid in hot paths.

Best Practices

- Prefer implicit conversions when they are guaranteed safe.
- Be explicit and intentional with narrowing conversions; validate ranges.
- Minimize boxing by using generics and avoiding APIs that require object.

Read More

- Microsoft Docs: Conversions in C#: https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/numeric-conversions (https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/numeric-conversions)
- Microsoft Docs: Boxing and Unboxing: https://learn.microsoft.com/dotnet/csharp/programming-guide/types/boxing-and-unboxing (https://learn.microsoft.com/dotnet/csharp/programming-guide/types/boxing-and-unboxing)

Namespaces in C#

Namespaces in C#

Purpose of Namespaces

Namespaces organize types and prevent naming collisions across libraries and projects.

Key Concepts

- Logical grouping: types with related purpose live together.
- Disambiguation: identical type names can coexist in different namespaces.
- Using directives: bring a namespace into scope to shorten type names.
- Aliases: assign a local alias to a type or namespace to avoid ambiguity.

Design Tips

- Mirror folder structure with namespaces for clarity.
- Use company/product root (e.g., Company.Product.Module).
- Avoid deep nesting unless it communicates meaningful boundaries.

Read More

- Microsoft Docs: Namespaces: https://learn.microsoft.com/dotnet/csharp/fundamentals/types/namespaces (https://learn.microsoft.com/dotnet/csharp/fundamentals/types/namespaces)

Branching in C# (if/else, switch)

Branching in C# (if/else, switch)

When to Branch

Branching selects different execution paths based on conditions.

if / else

- Evaluate a boolean condition to choose a path.
- Multiple else-if blocks support more than two choices.

switch

- Good for discrete choices based on a single value.
- Pattern matching expands switch power (types, ranges, conditions) while staying readable.

Best Practices

- Keep conditions simple and intention-revealing.
- Prefer switch for many discrete cases; avoid long if/else chains.
- Extract complex conditions into well-named helpers for readability and reuse.

Read More

- Microsoft Docs: if-else: https://learn.microsoft.com/dotnet/csharp/language-reference/statements/selection-statements (https://learn.microsoft.com/dotnet/csharp/language-reference/statements/selection-statements)

- Microsoft Docs: switch and pattern matching: https://learn.microsoft.com/dotnet/csharp/language-reference/operators/switch-expression (https://learn.microsoft.com/dotnet/csharp/language-reference/operators/switch-expression)

Looping in C# (for, while, foreach)

Looping in C# (for, while, foreach)

Why Loops

Loops repeat work over a sequence or until a condition changes.

for / while

- for: use when you control an index and have clear start/stop/step.
- while: use when you loop until a condition becomes false.

foreach

- Iterates elements of a collection in sequence order.
- Emphasizes the element rather than index bookkeeping.

Pitfalls and Tips

- Avoid off-by-one errors by defining inclusive/exclusive bounds explicitly.
- Ensure loop termination; mutate conditions correctly.
- Prefer foreach for readability when indexing isn't needed.

Read More

- Microsoft Docs: Iteration statements: https://learn.microsoft.com/dotnet/csharp/language-reference/statements/iteration-statements (https://learn.microsoft.com/dotnet/csharp/language-reference/statements/iteration-statements)

Common Language Runtime (CLR)

Common Language Runtime (CLR)

Role of CLR

- Executes .NET code (IL -> native) via Just-In-Time (JIT) compilation.
- Provides memory management (GC), type safety, security, and exceptions.

Key Services

- Garbage Collection (automatic memory reclamation)
- JIT Compilation and Tiered JIT
- Type System and Metadata
- Assemblies and AppDomains (historical) / AssemblyLoadContext (modern)

Why It Matters

Performance, safety, interoperability, and deployment behavior all depend on CLR services.

Read More

- https://learn.microsoft.com/dotnet/standard/clr (https://learn.microsoft.com/dotnet/standard/clr)

.NET Framework Class Library (BCL/FCL)

.NET Framework Class Library (BCL/FCL)

What It Is

The foundational library of types for collections, IO, networking, threading, etc.

Common Namespaces

- System, System.Collections.Generic
- System.IO, System.Net.Http
- System.Threading, System.Threading.Tasks

Tips

Prefer BCL types before third-party libraries; they're well-tested and supported.

Read More

- https://learn.microsoft.com/dotnet/standard/class-library-overview (https://learn.microsoft.com/dotnet/standard/class-library-overview)

IDE Setup (Visual Studio / VS Code)

IDE Setup (Visual Studio / VS Code)

VS Code

- Install C# Dev Kit extension (and dependencies).
- Use dotnet SDK (https://dotnet.microsoft.com/download (https://dotnet.microsoft.com/download)).

Visual Studio

- Install the ".NET desktop development" and ".NET Web" workloads as needed.

Tips

- Enable nullable reference types in projects for safer code.
- Use formatters and analyzers (EditorConfig, Roslyn analyzers).

Read More

- https://learn.microsoft.com/dotnet/core/tutorials/with-visual-studio (https://learn.microsoft.com/dotnet/core/tutorials/with-visual-studio)
- https://learn.microsoft.com/dotnet/core/tutorials/with-visual-studio-code (https://learn.microsoft.com/dotnet/core/tutorials/with-visual-studio-code)

Classes and Objects

Classes and Objects

Concepts

- Class: blueprint of state (fields/properties) and behavior (methods).
- Object: instance of a class with its own state.
- Constructors/Destructors: init/cleanup; destructors are rare in modern .NET.
- Properties/Indexers: encapsulate access; prefer auto-properties with validation as needed.
- Static Members: shared across all instances.

Read More

- https://learn.microsoft.com/dotnet/csharp/fundamentals/types/classes (https://learn.microsoft.com/dotnet/csharp/fundamentals/types/classes)

OOP Principles

OOP Principles

Encapsulation

Hide implementation details; expose a clear interface.

Inheritance

Share and specialize behavior; avoid deep hierarchies.

Polymorphism

Same interface, different implementations (overloading vs overriding).

Abstraction

Model only what’s essential using abstract classes or interfaces.

Read More

- https://learn.microsoft.com/dotnet/csharp/fundamentals/object-oriented/ (https://learn.microsoft.com/dotnet/csharp/fundamentals/object-oriented/)

Advanced OOP

Advanced OOP

Structs vs Classes

- Structs are value types (stack-friendly, small, immutable preferred).

Enums & Nested Types

- Enums for constrained sets; nested types for close coupling.

Partial Types

- Split type definitions across files for organization.

Read More

- https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/struct (https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/struct)
- https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/enum (https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/enum)

Built-in Collections

Built-in Collections

Core Types

- Arrays, List, Dictionary<TKey,TValue>
- HashSet, Queue, Stack
- Concurrent collections for multi-threading scenarios

Selection Tips

- List for ordered lists; Dictionary for key lookups; HashSet for uniqueness.

Read More

- https://learn.microsoft.com/dotnet/standard/collections/ (https://learn.microsoft.com/dotnet/standard/collections/)

Custom Collections

Custom Collections

Implementing IEnumerable

Expose sequences safely; favor yield for lazy iteration.

Implementing IList

Provide indexable, mutable collections when needed—consider complexity and invariants.

Read More

- https://learn.microsoft.com/dotnet/api/system.collections.generic.ienumerable-1 (https://learn.microsoft.com/dotnet/api/system.collections.generic.ienumerable-1)
- https://learn.microsoft.com/dotnet/api/system.collections.generic.ilist-1 (https://learn.microsoft.com/dotnet/api/system.collections.generic.ilist-1)

Exception Handling

Exception Handling

Principles

- Use exceptions for exceptional conditions, not flow control.
- try/catch/finally for handling and cleanup.

Guidance

- Catch only what you can handle.
- Preserve stack traces; include context in messages.

Read More

- https://learn.microsoft.com/dotnet/csharp/fundamentals/exceptions/ (https://learn.microsoft.com/dotnet/csharp/fundamentals/exceptions/)

Custom Exceptions

Custom Exceptions

When to Create One

- To represent domain-specific error conditions.

Best Practices

- Derive from Exception (or a relevant subclass).
- Be serializable; include useful constructors.

Read More

- https://learn.microsoft.com/dotnet/standard/exceptions/how-to-create-user-defined-exceptions (https://learn.microsoft.com/dotnet/standard/exceptions/how-to-create-user-defined-exceptions)

Debugging Techniques

Debugging Techniques

Tools

- Breakpoints, Watches, Locals, Call Stack, Immediate Window.
- Logging for post-mortem analysis.

Tips

- Reproduce reliably; binary search the fault; isolate changes.

Read More

- https://learn.microsoft.com/visualstudio/debugger/debugger-feature-tour (https://learn.microsoft.com/visualstudio/debugger/debugger-feature-tour)

Delegates and Events

Delegates and Events

Delegates

Type-safe function pointers; Action/Func cover common signatures.

Lambdas

Inline function expressions; capture variables (closures) with care.

Events

Publish/subscribe pattern for notifications.

Read More

- https://learn.microsoft.com/dotnet/csharp/programming-guide/delegates/ (https://learn.microsoft.com/dotnet/csharp/programming-guide/delegates/)
- https://learn.microsoft.com/dotnet/csharp/programming-guide/events/ (https://learn.microsoft.com/dotnet/csharp/programming-guide/events/)

LINQ

LINQ

Two Styles

- Query syntax vs method syntax; both compile to the same operators.

Key Concepts

- Deferred execution, filtering, projection, grouping, joining.

IQueryable vs IEnumerable

- IEnumerable executes in-memory; IQueryable can translate to remote providers.

Read More

- https://learn.microsoft.com/dotnet/csharp/programming-guide/concepts/linq/ (https://learn.microsoft.com/dotnet/csharp/programming-guide/concepts/linq/)

Asynchronous Programming

Asynchronous Programming

async/await

- Compose asynchronous operations without blocking threads.

Tasks and Parallelism

- Task represents ongoing work; Parallel APIs for data parallelism.

Cancellation Tokens

- Cooperative cancellation via tokens passed to async operations.

Read More

- https://learn.microsoft.com/dotnet/csharp/asynchronous-programming/ (https://learn.microsoft.com/dotnet/csharp/asynchronous-programming/)

ADO.NET

ADO.NET

Connected vs Disconnected

- Connected: direct commands and readers over open connections.
- Disconnected: DataSet/DataTable for offline manipulation.

Transactions

- Ensure atomicity across multiple operations.

Read More

- https://learn.microsoft.com/dotnet/framework/data/adonet/ado-net-overview (https://learn.microsoft.com/dotnet/framework/data/adonet/ado-net-overview)

Entity Framework Core

Entity Framework Core

Approaches

- Code-First vs Database-First; migrations manage schema changes.

Tips

- Use DbContext lifetime appropriately; track changes intentionally.

Read More

- https://learn.microsoft.com/ef/core/ (https://learn.microsoft.com/ef/core/)

File I/O

File I/O

Streams

- FileStream/MemoryStream are building blocks for reading/writing.

Serialization

- JSON, XML, Binary trade-offs: interoperability, performance, fidelity.

Read More

- https://learn.microsoft.com/dotnet/standard/io/ (https://learn.microsoft.com/dotnet/standard/io/)

WPF: XAML Basics

WPF: XAML Basics

Layouts

- Grid, StackPanel for arranging controls.

Controls

- Buttons, TextBox, and common controls.

Data Binding

- INotifyPropertyChanged for reactive UI.

Read More

- https://learn.microsoft.com/dotnet/desktop/wpf/xaml-services/?view=netdesktop-8.0 (https://learn.microsoft.com/dotnet/desktop/wpf/xaml-services/?view=netdesktop-8.0)

WPF: Advanced

WPF: Advanced

Styles and Templates

- Separate look from behavior; reuse visuals.

Commands

- ICommand to decouple UI actions from handlers.

MVVM Pattern

- Model-View-ViewModel for testable, maintainable UI.

Read More

- https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-studio (https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-studio)

ASP.NET Core Fundamentals

ASP.NET Core Fundamentals

Middleware Pipeline

- Request flows through configurable components.

Razor Pages vs MVC

- Pages for page-centric apps; MVC for controllers/views.

Web API

- Build RESTful endpoints; content negotiation and model binding.

Read More

- https://learn.microsoft.com/aspnet/core/ (https://learn.microsoft.com/aspnet/core/)

Blazor

Blazor

Components and Binding

- Reusable components; bind to data and events.

Hosting Models

- Server vs WebAssembly; trade-offs in latency and capabilities.

Dependency Injection

- Built-in DI for services and state.

Read More

- https://learn.microsoft.com/aspnet/core/blazor/ (https://learn.microsoft.com/aspnet/core/blazor/)

Web Security

Web Security

Authentication

- Cookies, JWT, external providers.

Authorization

- Role- and policy-based strategies.

Transport & Cross-Origin

- HTTPS everywhere; CORS for cross-origin requests.

Read More

- https://learn.microsoft.com/aspnet/core/security/ (https://learn.microsoft.com/aspnet/core/security/)

Xamarin.Forms

Xamarin.Forms

XAML Layouts

- Build UI using XAML markup and controls.

Navigation

- Master-Detail, Tabs, and navigation stacks.

Read More

- https://learn.microsoft.com/xamarin/xamarin-forms/ (https://learn.microsoft.com/xamarin/xamarin-forms/)

Mobile Features

Mobile Features

Local Storage

- SQLite.NET for on-device data.

Platform-Specific Code

- DependencyService/Handlers for native features.

OAuth 2.0

- Secure auth with external identity providers.

Read More

- https://learn.microsoft.com/xamarin/ (https://learn.microsoft.com/xamarin/)

Cloud Deployment

Cloud Deployment

Options

- Azure App Service, Docker containers, AWS Elastic Beanstalk.

Considerations

- Configuration, secrets, logging, scaling.

Read More

- https://learn.microsoft.com/azure/app-service/ (https://learn.microsoft.com/azure/app-service/)
- https://docs.docker.com/get-started/ (https://docs.docker.com/get-started/)

CI/CD Pipelines

CI/CD Pipelines

Tools

- GitHub Actions, Azure DevOps Pipelines.

Practices

- Build/test on each push; versioning; automated deployments.

Read More

- https://learn.microsoft.com/azure/devops/pipelines/ (https://learn.microsoft.com/azure/devops/pipelines/)
- https://docs.github.com/actions (https://docs.github.com/actions)