C# Revision - Day 1 Fundamentals

1. Basic Syntax and Structure

Program Structure

```
using System;  // Import namespace

namespace MyFirstProgram  // Declare namespace
{
    class Program  // Declare class
    {
        static void Main(string[] args)  // Main method - entry point
        {
            Console.WriteLine("Hello, World!");  // Statement
        }
    }
}
```

Key Components:

- **Namespaces**: Containers for organizing code and preventing naming conflicts
- **Classes**: Templates for creating objects, containing methods and data
- **Methods**: Named blocks of code that perform specific tasks
- **Main Method**: Entry point of execution in console applications
- **Statements**: Instructions that perform actions, ending with semicolons

C# is:

- **Strongly typed**: Variables must be declared with a specific type
- **Object-oriented**: Based on classes and objects
- **Component-oriented**: Supports components with properties and events
- **Type-safe**: Types are checked during compilation to prevent errors
- **Garbage collected**: Memory management is automatic

2. Data Types and Variables

Value Types (Stored on the stack)

Integral Types

```
byte b = 255;           // 8 bits, 0 to 255
sbyte sb = -128;        // 8 bits, -128 to 127
short s = 32767;        // 16 bits, -32,768 to 32,767
```

```
ushort us = 65535;      // 16 bits, 0 to 65,535
int i = 2147483647;     // 32 bits, -2,147,483,648 to 2,147,483,647
uint ui = 4294967295;   // 32 bits, 0 to 4,294,967,295
long l = 9223372036854775807;  // 64 bits, very large range
ulong ul = 18446744073709551615;  // 64 bits, very large positive range
```

Floating Point Types

```
float f = 3.14f;        // 32 bits, ~7 digits precision, requires 'f' suffix
double d = 3.14159;     // 64 bits, ~15-16 digits precision
decimal m = 3.14159m;   // 128 bits, 28-29 significant digits, requires 'm' suffix
```

Other Value Types

```
bool isValid = true;    // Boolean (true/false)
char c = 'A';           // 16-bit Unicode character
```

Reference Types (Stored on the heap, reference on stack)

```
string name = "John";   // String (immutable sequence of characters)
object obj = new object(); // Base type of all types
dynamic dyn = 100;      // Type determined at runtime
```

Nullable Types

Allows value types to also hold null:

```
int? nullableInt = null;
bool? nullableBool = null;
```

Variable Declaration and Initialization

```
// Declaration then initialization
int age;
age = 25;

// Combined declaration and initialization
string name = "John";

// Type inference with 'var'
var message = "Hello";  // Compiler infers this is a string
var count = 10;         // Compiler infers this is an int
```

Constants

Values that cannot be changed after declaration:

```
const double Pi = 3.14159;
const string AppName = "My C# App";
```

Theory: Value vs. Reference Types

- **Value types** store their data directly and each variable has its own copy
- **Reference types** store a reference (memory address) to the actual data
- When you assign a value type to another variable, it creates a copy
- When you assign a reference type, both variables reference the same object

3. Type Conversion

Implicit Conversion (Widening)

Occurs when conversion is safe and no data loss can occur:

```
int i = 100;
long l = i;      // int fits in long (safe)
float f = l;     // long fits in float (safe but potential precision loss)
```

Explicit Conversion (Casting)

Required when conversion might cause data loss:

```
double d = 123.45;
int i = (int)d;         // Explicitly cast, decimal part is truncated
```

Helper Methods

For more controlled conversions:

```
// Using Convert class
int i = Convert.ToInt32("123");
bool b = Convert.ToBoolean("True");
double d = Convert.ToDouble("123.45");

// Using Parse methods
int i2 = int.Parse("123");
double d2 = double.Parse("123.45");

// Using TryParse for safer conversion
bool success = int.TryParse("123", out int result);
if (success)
```

```
{
    // Use result
}
```

Type Testing

Test an object's type:

```
object obj = "Hello";
bool isString = obj is string;        // true
bool isInt = obj is int;              // false

// Type pattern matching
if (obj is string message)
{
    // message is already cast to string
    Console.WriteLine(message.Length);
}
```

4. Operators

Arithmetic Operators

```
int a = 10, b = 3;
int sum = a + b;        // 13 (addition)
int diff = a - b;       // 7 (subtraction)
int product = a * b;    // 30 (multiplication)
int quotient = a / b;   // 3 (integer division)
int remainder = a % b;  // 1 (modulus - remainder after division)

// Increment/decrement
int x = 5;
x++;                    // Post-increment (use x then increment)
++x;                    // Pre-increment (increment x then use it)
x--;                   // Post-decrement (use x then decrement)
--x;                   // Pre-decrement (decrement x then use it)
```

Comparison Operators

```
bool isEqual = (a == b);        // Equal to
bool isNotEqual = (a != b);     // Not equal to
bool isGreater = (a > b);       // Greater than
bool isLess = (a < b);          // Less than
bool isGreaterOrEqual = (a >= b); // Greater than or equal to
bool isLessOrEqual = (a <= b);  // Less than or equal to
```

Logical Operators

```
bool condition1 = true, condition2 = false;

bool andResult = condition1 && condition2;  // Logical AND (both must be true)
bool orResult = condition1 || condition2;   // Logical OR (at least one must be true)
bool notResult = !condition1;               // Logical NOT (inverts the boolean)

// Short-circuit evaluation:
// && stops evaluating if the first operand is false
// || stops evaluating if the first operand is true
```

Assignment Operators

```
int value = 10;
value += 5;      // value = value + 5
value -= 3;      // value = value - 3
value *= 2;      // value = value * 2
value /= 4;      // value = value / 4
value %= 3;      // value = value % 3
```

Null-Related Operators

```
// Null-conditional operator (?.)
string name = null;
int? length = name?.Length;  // Null if name is null, otherwise returns length

// Null-coalescing operator (??)
string displayName = name ?? "Anonymous";  // Use "Anonymous" if name is null

// Null-coalescing assignment (??=)
string username = null;
username ??= "Guest";  // Assigns "Guest" to username only if username is null
```

Ternary Conditional Operator

Shorthand for if-else:

```
int age = 20;
string message = (age >= 18) ? "Adult" : "Minor";
```

5. Control Flow Statements

If-Else Statements

```
int hour = 14;

if (hour < 12)
{
    Console.WriteLine("Good morning!");
}
else if (hour < 18)
{
    Console.WriteLine("Good afternoon!");
}
else
{
    Console.WriteLine("Good evening!");
}

// Single-line if (braces optional but recommended)
if (hour < 6) Console.WriteLine("It's early!");
```

Switch Statement

```
int day = 3;
string dayName;

switch (day)
{
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
    case 7:
        dayName = "Sunday";
        break;
    default:
        dayName = "Invalid day";
        break;
}
```

Switch Expression (C# 8.0+)

Modern alternative to switch statement:

```
string dayName = day switch
{
    1 => "Monday",
    2 => "Tuesday",
    3 => "Wednesday",
    4 => "Thursday",
    5 => "Friday",
    6 => "Saturday",
    7 => "Sunday",
    _ => "Invalid day"  // Default case
};
```

Pattern Matching in Switch

```
object obj = "Hello";

switch (obj)
{
    case string s when s.Length > 5:
        Console.WriteLine("Long string");
        break;
    case string s:
        Console.WriteLine("Short string");
        break;
    case int i:
        Console.WriteLine("Integer");
        break;
    default:
        Console.WriteLine("Something else");
        break;
}
```

6. Loops

For Loop

Use when you know the number of iterations in advance:

```
// Basic for loop (print numbers 0-4)
for (int i = 0; i < 5; i++)
{
   Console.WriteLine(i);
}

// Decrementing for loop
for (int i = 10; i > 0; i--)
{
   Console.WriteLine(i);
}

// Multiple control variables
for (int i = 0, j = 10; i < j; i++, j--)
{
   Console.WriteLine($"i = {i}, j = {j}");
}
```

While Loop

Executes as long as a condition is true:

```
int count = 0;
while (count < 5)
{
   Console.WriteLine(count);
   count++;
}
```

Do-While Loop

Similar to while, but always executes at least once:

```
int num = 0;
do
{
   Console.WriteLine(num);
   num++;
} while (num < 5);
```

Foreach Loop

Used to iterate through collections:

```
string[] colors = { "Red", "Green", "Blue" };
foreach (string color in colors)
{
    Console.WriteLine(color);
}
```

Jump Statements

Control the flow within loops:

```
// Break - exits the loop completely
for (int i = 0; i < 10; i++)
{
    if (i == 5) break;  // Exit when i reaches 5
    Console.WriteLine(i);
}

// Continue - skips the current iteration
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 0) continue;  // Skip even numbers
    Console.WriteLine(i);
}

// Return - exits the current method
bool ContainsValue(int[] numbers, int value)
{
    foreach (int num in numbers)
    {
        if (num == value) return true;
    }
    return false;
}
```

7. Arrays and Collections

Arrays

Fixed-size collections of same-type elements:

```
// Declaration and initialization
int[] numbers = new int[5];  // Array of 5 integers (default values: 0)
numbers[0] = 10;             // Assign value to first element
numbers[1] = 20;
```

```
// Shorthand initialization
string[] fruits = { "Apple", "Banana", "Cherry" };

// Multi-dimensional arrays
int[,] grid = new int[3, 2];  // 3 rows, 2 columns
grid[0, 0] = 1;
grid[0, 1] = 2;

// Jagged arrays (arrays of arrays)
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[] { 1, 2, 3 };
jaggedArray[1] = new int[] { 4, 5 };
jaggedArray[2] = new int[] { 6, 7, 8, 9 };

// Array methods
Array.Sort(numbers);           // Sort array
Array.Reverse(numbers);        // Reverse array
int index = Array.IndexOf(fruits, "Banana");  // Find index of element
bool exists = Array.Exists(numbers, n => n > 15);  // Check if any element matches
```

Lists

Resizable collections from System.Collections.Generic:

```
using System.Collections.Generic;

// Create a list
List<string> names = new List<string>();

// Add elements
names.Add("Alice");
names.Add("Bob");
names.Add("Charlie");

// Initialize with values
List<int> scores = new List<int> { 90, 85, 77, 92 };

// Access elements
string firstPerson = names[0];  // "Alice"

// Insert at specific position
names.Insert(1, "David");  // Insert between Alice and Bob

// Find elements
bool hasAlice = names.Contains("Alice");  // true
int bobIndex = names.IndexOf("Bob");      // now 2 after insert
```

```
// Remove elements
names.Remove("Charlie");  // Remove by value
names.RemoveAt(0);        // Remove by index (Alice)

// Find elements with predicate
List<int> highScores = scores.FindAll(s => s >= 90);  // [90, 92]
int firstHighScore = scores.Find(s => s >= 90);       // 90

// Sort list
scores.Sort();  // [77, 85, 90, 92]

// Get count
int count = names.Count;
```

Dictionary

Collection of key-value pairs:

```
// Create a dictionary
Dictionary<string, int> ages = new Dictionary<string, int>();

// Add elements
ages.Add("Alice", 25);
ages.Add("Bob", 30);
ages["Charlie"] = 35;  // Alternative syntax

// Access by key
int bobAge = ages["Bob"];  // 30

// Check if key exists
if (ages.ContainsKey("David"))
{
    // Do something
}

// Safe access with TryGetValue
if (ages.TryGetValue("Alice", out int aliceAge))
{
    Console.WriteLine($"Alice's age: {aliceAge}");
}

// Iterate through dictionary
foreach (var pair in ages)
{
    Console.WriteLine($"{pair.Key} is {pair.Value} years old");
}
```

```
// Get keys and values
Dictionary<string, int>.KeyCollection keys = ages.Keys;
Dictionary<string, int>.ValueCollection values = ages.Values;
```

HashSet

Collection of unique elements:

```
HashSet<int> uniqueNumbers = new HashSet<int>();
uniqueNumbers.Add(1);
uniqueNumbers.Add(2);
uniqueNumbers.Add(1);  // Duplicate, won't be added

// Check membership
bool contains3 = uniqueNumbers.Contains(3);  // false

// Set operations
HashSet<int> setA = new HashSet<int> { 1, 2, 3 };
HashSet<int> setB = new HashSet<int> { 2, 3, 4 };

setA.UnionWith(setB);       // A becomes { 1, 2, 3, 4 }
setA.IntersectWith(setB);   // A becomes { 2, 3, 4 }
setA.ExceptWith(setB);      // A becomes { } (empty)
```

8. String Manipulation

String Creation and Basic Operations

```
// String creation
string firstName = "John";
string lastName = "Doe";

// Concatenation
string fullName = firstName + " " + lastName;  // "John Doe"

// String interpolation (C# 6.0+) - preferred
string greeting = $"Hello, {firstName} {lastName}!";  // "Hello, John Doe!"

// String.Format (older method)
string formatted = String.Format("Hello, {0} {1}!", firstName, lastName);

// String properties
int length = firstName.Length;  // 4
bool isEmpty = string.IsNullOrEmpty(firstName);  // false
bool isWhitespace = string.IsNullOrWhiteSpace("   ");  // true
```

String Methods

```
string message = "Hello, World!";

// Case conversion
string upper = message.ToUpper();  // "HELLO, WORLD!"
string lower = message.ToLower();  // "hello, world!"

// Checking content
bool startsWithHello = message.StartsWith("Hello");  // true
bool endsWithWorld = message.EndsWith("World!");     // true
bool containsComma = message.Contains(",");          // true

// Finding position
int commaPos = message.IndexOf(",");  // 5
int notFound = message.IndexOf("x");  // -1 (not found)
int lastO = message.LastIndexOf("o"); // 8 (position of 'o' in "World")

// Substring
string part = message.Substring(7, 5);  // "World"
string fromPos = message.Substring(7);  // "World!" (from index 7 to end)

// Trimming whitespace
string padded = "  text  ";
string trimmed = padded.Trim();      // "text"
string trimStart = padded.TrimStart();  // "text  "
string trimEnd = padded.TrimEnd();    // "  text"

// Replacing text
string newMessage = message.Replace("World", "C#");  // "Hello, C#!"
string noCommas = message.Replace(",", "");          // "Hello World!"
```

String Comparison

```
string str1 = "apple";
string str2 = "Apple";

// Case-sensitive comparison (default)
bool areEqual = str1 == str2;  // false
bool areEqual2 = str1.Equals(str2);  // false

// Case-insensitive comparison
bool areEqualIgnoreCase = str1.Equals(str2, StringComparison.OrdinalIgnoreCase);  // true

// Comparison for sorting
int compareResult = string.Compare(str1, str2);  // > 0 (str1 comes after str2 in sort order)
int compareIgnoreCase = string.Compare(str1, str2, ignoreCase: true);  // 0 (equal when ignoring case)
```

String Splitting and Joining

```
// Splitting
string csvData = "apple,orange,banana";
string[] fruits = csvData.Split(',');  // ["apple", "orange", "banana"]

string data = "one|two|three";
string[] parts = data.Split('|');  // ["one", "two", "three"]

string text = "The quick brown fox";
string[] words = text.Split(' ');  // ["The", "quick", "brown", "fox"]

// Joining
string[] colors = { "red", "green", "blue" };
string colorList = string.Join(", ", colors);  // "red, green, blue"
string dashSeparated = string.Join("-", words);  // "The-quick-brown-fox"
```

StringBuilder

For efficiently building strings with many modifications:

```
using System.Text;

StringBuilder builder = new StringBuilder();
builder.Append("Hello");
builder.Append(" ");
builder.Append("World");
builder.AppendLine("!");  // Adds newline after
builder.AppendLine("Multiple lines");

// Insert at specific position
builder.Insert(5, " beautiful");  // "Hello beautiful World!"

// Replace part of string
builder.Replace("World", "C#");

// Remove part of string
builder.Remove(6, 10);  // Remove 10 chars starting at index 6

// Final result
string result = builder.ToString();
```

9. DateTime and TimeSpan

DateTime Basics

```
// Current date and time
DateTime now = DateTime.Now;       // Local time
DateTime utcNow = DateTime.UtcNow; // UTC time
DateTime today = DateTime.Today;   // Today's date with time set to 00:00:00

// Creating specific dates
DateTime birthdate = new DateTime(1990, 5, 15);  // May 15, 1990
DateTime christmas = new DateTime(2023, 12, 25, 0, 0, 0);  // With time component

// Parsing from string
DateTime parsed = DateTime.Parse("2023-04-15");  // Depends on culture
DateTime exactParsed = DateTime.ParseExact("15/04/2023", "dd/MM/yyyy", null);

// Getting components
int year = now.Year;
int month = now.Month;
int day = now.Day;
int hour = now.Hour;
int minute = now.Minute;
int second = now.Second;
int millisecond = now.Millisecond;

// Day of week
DayOfWeek dayOfWeek = now.DayOfWeek;  // e.g., Monday, Tuesday, etc.
bool isWeekend = dayOfWeek == DayOfWeek.Saturday || dayOfWeek ==
DayOfWeek.Sunday;

// String representation
string fullDate = now.ToString();  // Default format
string shortDate = now.ToShortDateString();  // e.g., 4/23/2025
string longDate = now.ToLongDateString();    // e.g., Wednesday, April 23, 2025
string shortTime = now.ToShortTimeString();  // e.g., 3:42 PM
string longTime = now.ToLongTimeString();    // e.g., 3:42:15 PM
string custom = now.ToString("yyyy-MM-dd HH:mm:ss");  // Custom format
```

DateTime Operations

```
DateTime today = DateTime.Today;

// Adding/subtracting time
DateTime tomorrow = today.AddDays(1);
DateTime nextWeek = today.AddDays(7);
DateTime nextMonth = today.AddMonths(1);
DateTime nextYear = today.AddYears(1);
DateTime twoHoursLater = now.AddHours(2);
DateTime thirtyMinutesBefore = now.AddMinutes(-30);
```

```
// First/last day of month
DateTime firstDayOfMonth = new DateTime(today.Year, today.Month, 1);
DateTime lastDayOfMonth = firstDayOfMonth.AddMonths(1).AddDays(-1);

// Difference between dates
TimeSpan age = today - birthdate;
int daysDifference = age.Days;
double yearsDifference = age.TotalDays / 365.25;  // Approximate years

// Comparing dates
bool isBefore = birthdate < today;  // true
bool isAfter = birthdate > today;   // false
bool isSameDay = today.Date == tomorrow.Date;  // false
```

TimeSpan

Represents a time interval:

```
// Creating a TimeSpan
TimeSpan oneHour = TimeSpan.FromHours(1);
TimeSpan twoMinutes = TimeSpan.FromMinutes(2);
TimeSpan threeSeconds = TimeSpan.FromSeconds(3);
TimeSpan customTime = new TimeSpan(1, 30, 45);  // 1 hour, 30 minutes, 45 seconds
TimeSpan fullCustom = new TimeSpan(2, 3, 30, 45, 500);  // 2 days, 3 hours, 30 min, 45 sec, 500 ms

// TimeSpan properties
int days = customTime.Days;          // 0
int hours = customTime.Hours;        // 1
int minutes = customTime.Minutes;    // 30
int seconds = customTime.Seconds;    // 45
int milliseconds = customTime.Milliseconds;  // 0

// Total elapsed time in different units
double totalHours = customTime.TotalHours;        // 1.5125
double totalMinutes = customTime.TotalMinutes;    // 90.75
double totalSeconds = customTime.TotalSeconds;    // 5445

// TimeSpan arithmetic
TimeSpan sum = oneHour + twoMinutes;              // 1 hour 2 minutes
TimeSpan difference = customTime - oneHour;       // 30 minutes 45 seconds
TimeSpan doubled = customTime * 2;                // 3 hours 1 minute 30 seconds
TimeSpan halved = customTime / 2;                 // 45 minutes 22.5 seconds

// String representation
string timeString = customTime.ToString();  // "01:30:45"
```

10. Basic File Operations

Reading from Files

```
using System.IO;

// Read all text at once
string content = File.ReadAllText("file.txt");

// Read all lines into a string array
string[] lines = File.ReadAllLines("file.txt");

// Read file line by line (more memory efficient for large files)
foreach (string line in File.ReadLines("file.txt"))
{
    Console.WriteLine(line);
}

// Using StreamReader
using (StreamReader reader = new StreamReader("file.txt"))
{
    while (!reader.EndOfStream)
    {
        string line = reader.ReadLine();
        Console.WriteLine(line);
    }
}

// Reading binary data
byte[] data = File.ReadAllBytes("data.bin");
```

Writing to Files

```
// Write all text at once
File.WriteAllText("output.txt", "Hello, World!");

// Write multiple lines at once
string[] newLines = { "Line 1", "Line 2", "Line 3" };
File.WriteAllLines("output.txt", newLines);

// Using StreamWriter
using (StreamWriter writer = new StreamWriter("output.txt"))
{
    writer.WriteLine("First line");
    writer.WriteLine("Second line");
    writer.Write("No newline after this");
    writer.WriteLine("This is on the same line");
}

// Append to file
File.AppendAllText("log.txt", "New log entry\n");

// Using StreamWriter to append
using (StreamWriter writer = new StreamWriter("log.txt", append: true))
{
    writer.WriteLine("Appended line");
}

// Writing binary data
byte[] data = { 0x48, 0x65, 0x6C, 0x6C, 0x6F };  // "Hello" in ASCII
File.WriteAllBytes("data.bin", data);
```

File and Directory Operations

```
// Check if file exists
bool exists = File.Exists("file.txt");

// Copy file
File.Copy("source.txt", "destination.txt", overwrite: true);

// Move/rename file
File.Move("old.txt", "new.txt");

// Delete file
File.Delete("temp.txt");

// File information
FileInfo fileInfo = new FileInfo("file.txt");
long size = fileInfo.Length;  // File size in bytes
DateTime created = fileInfo.CreationTime;
DateTime modified = fileInfo.LastWriteTime;
string extension = fileInfo.Extension;

// Directory operations
bool dirExists = Directory.Exists("logs");
Directory.CreateDirectory("logs");
Directory.Delete("temp", recursive: true);  // Delete directory and contents
Directory.Move("oldDir", "newDir");

// Get files and directories
string[] files = Directory.GetFiles(".", "*.txt");  // Get all .txt files
string[] directories = Directory.GetDirectories(".");  // Get all subdirectories
```

```
string[] allFiles = Directory.GetFiles(".", "*.*", SearchOption.AllDirectories);  // All files including subdirectories
```

File Paths

```
// Working with paths
string fileName = "document.txt";
string directory = @"C:\Users\Documents";
string fullPath = Path.Combine(directory, fileName);  // C:\Users\Documents\document.txt

// Path components
string extension = Path.GetExtension(fullPath);       // .txt
string fileNameOnly = Path.GetFileName(fullPath);     // document.txt
string fileNameNoExt = Path.GetFileNameWithoutExtension(fullPath);  // document
string directoryName = Path.GetDirectoryName(fullPath);  // C:\Users\Documents

// Special directories
string tempPath = Path.GetTempPath();
string randomFileName = Path.GetRandomFileName();
string tempFileName = Path.GetTempFileName();
```

Theoretical Concepts

Memory Management in C#

C# uses automatic memory management through garbage collection:

- **Stack**: Stores value types and references (addresses) to objects on the heap
- **Heap**: Stores objects (reference types like strings, arrays, classes)
- **Garbage Collector**: Automatically frees memory that's no longer referenced
- **Disposable Pattern**: Using the `IDisposable` interface and `using` statements for managing unmanaged resources

Naming Conventions

- **PascalCase**: For class names, method names, and public members (`public void CalculateTotal()`)
- **camelCase**: For local variables and method parameters (`int totalAmount`)
- **_camelCase**: For private fields (`private int _count`)
- **ALL_CAPS**: For constants (`const int MAX_SIZE = 100`)

Best Practices for Day 1 Concepts

1. **Use appropriate data types**: Choose the most appropriate type for the data (e.g., decimal for money)

2. **Prefer string interpolation**: Use `$"Hello {name}"` instead of string concatenation
3. **Use StringBuilder** for string concatenation in loops
4. **Initialize variables** when declaring them
5. **Use proper exception handling** around file operations
6. **Use path combiners** instead of string concatenation for file paths
7. **Prefer foreach** when you don't need the index
8. **Use descriptive variable names** that indicate purpose
9. **Avoid magic numbers**: Use constants or named variables
10. **Use var judiciously**: Good for obvious types, avoid when type is not clear