C# Day 1 - Theoretical Concepts

Basic C# Concepts

What is C#?

C# is a modern, type-safe, object-oriented programming language developed by Microsoft as part of the .NET initiative. It was designed by Anders Hejlsberg and his team and was first released in 2000. C# combines the power and flexibility of C++ with the simplicity of Visual Basic.

Key Characteristics of C#

- **Type-safe**: The compiler ensures type-correctness, preventing type errors at runtime
- **Object-oriented**: Based on classes and objects, supporting encapsulation, inheritance, and polymorphism
- **Component-oriented**: Supports components with properties and events
- **Strongly typed**: Variables must be declared with a specific type
- **Garbage collected**: Memory management is automatic
- **Modern**: Supports features like generics, LINQ, async/await, pattern matching, etc.

C# and .NET

C# code runs within the .NET environment:

- **CLR (Common Language Runtime)**: The execution environment responsible for running .NET applications
- **FCL (Framework Class Library)**: A comprehensive library of reusable classes and methods
- **CIL (Common Intermediate Language)**: The intermediate language that C# code is compiled into before being converted to machine code

Compilation Process

1. **Write** C# source code (.cs files)
2. **Compile** to CIL (previously called MSIL) using the C# compiler
3. **Package** code into assemblies (.dll or .exe)
4. **Execute** via JIT (Just-In-Time) compilation by the CLR

Data Types and Variables

Type System

C# has a unified type system where all types (including primitives) inherit from the `System.Object` class.

Value Types vs. Reference Types

- **Value Types**: Store actual data directly, allocated on the stack
    - Examples: int, float, bool, char, structs, enums
    - When assigned, values are copied
    - Memory released when they go out of scope
- **Reference Types**: Store a reference (address) to data on the heap
    - Examples: string, arrays, classes, interfaces
    - When assigned, the reference is copied, not the data
    - Memory managed by the garbage collector

Stack vs. Heap

- **Stack**:
    - Fast, fixed-size memory allocation
    - LIFO (Last-In-First-Out) data structure
    - Stores value types and references (addresses)
    - Automatically managed by the runtime
- **Heap**:
    - Dynamically allocated memory
    - Used for reference types and boxing value types
    - Managed by the garbage collector
    - More flexible but slightly slower

Variable Lifetimes

- **Local variables**: Exist from declaration to the end of the block
- **Method parameters**: Exist during method execution
- **Instance variables**: Exist as long as the containing object exists
- **Static variables**: Exist from program start until program end

Type Safety

C# is a type-safe language, which provides several benefits:

- Prevents operations from being performed on incompatible types
- Catches errors at compile time rather than runtime
- Provides clear contracts for methods and functions
- Enables IntelliSense features in the IDE

Boxing and Unboxing

1. **Boxing**: Converting a value type to a reference type (object)

    ```
    int i = 123;
    ```

2. object o = i; // Boxing
-
3. **Unboxing**: Converting a reference type back to a value type

   int j = (int)o; // Unboxing
-
- Boxing/unboxing has performance implications and should be minimized

Operators and Expressions

Operator Precedence

Operators in C# follow a specific order of precedence:

1. Primary operators (., [], (), etc.)
2. Unary operators (++, --, !, ~, etc.)
3. Multiplicative operators (*, /, %)
4. Additive operators (+, -)
5. Shift operators (<<, >>)
6. Relational operators (<, >, <=, >=)
7. Equality operators (==, !=)
8. Logical operators (&, ^, |, &&, ||)
9. Conditional operators (?:)
10. Assignment operators (=, +=, -=, etc.)

Short-Circuit Evaluation

Logical operators `&&` and `||` use short-circuit evaluation:

- `&&`: If the first operand is false, the second operand is not evaluated
- `||`: If the first operand is true, the second operand is not evaluated

Type Conversion Theory

Implicit Conversions

- No data loss possible
- No special syntax required
- Examples: int → long, float → double
- Occurs when target type can represent all possible values of source type

Explicit Conversions (casting)

- Potential data loss
- Requires explicit cast operator
- Examples: double → int (loses decimal part), long → int (might truncate)

- Required when compiler cannot guarantee safe conversion

User-Defined Conversions

- Classes and structs can define custom conversion operators
- Enables natural conversion syntax between custom types

Control Flow

How Control Flow Works

Control flow determines the order in which statements are executed:

- **Sequential execution**: Default top-to-bottom execution
- **Conditional execution**: Based on conditions (if, switch)
- **Iterative execution**: Repeating blocks (loops)
- **Jump statements**: Altering normal flow (break, continue, return)

Decision-Making Internals

- Conditional statements (if, switch) generate different IL code paths
- The runtime evaluates conditions and branches to appropriate code paths
- Pattern matching extends this with more sophisticated type and data matching

Switch Internals

- Simple switch statements with integer values may compile to jump tables
- More complex switches use series of conditional checks
- Switch expressions (C# 8.0+) provide more concise, expression-based syntax

Arrays and Collections

Arrays

- Contiguous block of memory with elements of the same type
- Fixed size once created
- Zero-indexed (first element is at index 0)
- Multidimensional arrays can be rectangular or jagged

Collection Interfaces

- **IEnumerable/IEnumerable<T>**: Base interface for all collections, enables foreach loops
- **ICollection<T>**: Adds methods for adding, removing, and counting elements
- **IList<T>**: Adds indexed access and more specific operations
- **IDictionary<TKey, TValue>**: Maps keys to values

Collections Performance Characteristics

- **List<T>**: Fast access by index, slower insertions/deletions in the middle
- **Dictionary<K,V>**: Fast lookups by key (near constant time), more memory usage
- **HashSet<T>**: Fast lookups and uniqueness checking
- **LinkedList<T>**: Fast insertions/deletions anywhere, slower access by index

Generics vs. Non-Generic Collections

- **Generic Collections**: Type-safe, better performance, no boxing/unboxing
- **Non-Generic Collections**: Store objects, require casting, more prone to errors

String Theory

String Immutability

- Strings in C# are immutable (cannot be changed after creation)
- When modifying a string, a new string is created
- Benefits: thread safety, security, hashcode caching
- Drawback: performance cost for many modifications (use StringBuilder instead)

String Intern Pool

- The CLR maintains a pool of unique string literals
- String literals with the same content refer to the same memory location
- Reduces memory usage for duplicate strings
- Can be manually controlled with String.Intern method

Character Encoding

- Strings in C# use UTF-16 encoding internally
- Each char is a 16-bit Unicode code unit
- Some Unicode characters (surrogate pairs) require two char units

String vs. StringBuilder

- **String**: Immutable, better for few concatenations
- **StringBuilder**: Mutable, better for many modifications
- StringBuilder maintains a buffer that can grow as needed

DateTime and TimeSpan

DateTime Representation

- DateTime in .NET represents a point in time
- Stored as 8-byte value (ticks since January 1, 0001)

- One tick = 100 nanoseconds
- Range: January 1, 0001 to December 31, 9999

TimeZone Considerations

- DateTime.Kind property indicates Local, UTC, or Unspecified
- DateTimeOffset includes timezone offset information
- Best practice: store times in UTC, convert to local only for display

DateTime vs. DateTimeOffset vs. TimeSpan

- **DateTime**: Point in time without timezone clarity
- **DateTimeOffset**: Point in time with timezone offset
- **TimeSpan**: Duration/time interval, not a specific point in time

File System Theory

Streams

- Abstract interface for reading/writing sequential data
- Types include FileStream, MemoryStream, NetworkStream
- Can be wrapped in other streams (BufferedStream, GZipStream, etc.)

Synchronous vs. Asynchronous I/O

- **Synchronous**: Blocks thread until operation completes
- **Asynchronous**: Returns control to caller while operation proceeds
- Async I/O recommended for UI applications and high-concurrency scenarios

File System Security

- .NET provides access control mechanisms
- Operations throw exceptions when permissions are insufficient
- Security attributes can be managed programmatically

Buffering

- Buffers improve I/O performance by reducing system calls
- StreamReader/StreamWriter provide buffering automatically
- Custom buffer sizes can be specified for performance tuning

Memory Management

Garbage Collection

- Automatic memory management system in .NET

- Tracks object references and reclaims memory from unreachable objects
- Uses generational approach (Gen 0, Gen 1, Gen 2)
- Compacts memory to reduce fragmentation

Garbage Collection Phases

1. **Marking**: Identify live objects
2. **Relocating**: Compact memory by moving objects
3. **Sweeping**: Reclaim memory from dead objects

Deterministic Cleanup with IDisposable

- **IDisposable**: Interface for freeing unmanaged resources
- **using statement**: Ensures Dispose is called even if exceptions occur
- Important for resources like files, database connections, network connections

Coding Standards and Best Practices

Naming Conventions

- **PascalCase**: Classes, methods, properties, namespaces
- **camelCase**: Local variables, method parameters
- **_camelCase**: Private fields
- **ALL_CAPS**: Constants

Code Organization

- One main class per file (filename matches class name)
- Related classes in the same namespace
- Logical organization of methods within a class
- Keep methods focused and short (single responsibility)

Code Quality Guidelines

- Write clear, self-documenting code
- Use meaningful names that express intent
- Follow consistent formatting
- Add comments for complex logic, not obvious code
- Write unit tests for critical code

Performance Considerations

- Minimize boxing and unboxing
- Use StringBuilder for complex string operations
- Dispose of unmanaged resources promptly
- Be careful with LINQ in performance-critical paths
- Consider struct vs class based on usage patterns