Comprehensive C# Interview Guide with Answers and Code Examples

Table of Contents

Basic C# Concepts

Q1. What are the different data types in C#?

Answer: C# has two main categories of data types:

Value Types:

- Primitive types: `int`, `float`, `double`, `char`, `bool`, `decimal`
- Structs and Enums

Reference Types:

- Classes, Interfaces, Arrays, Delegates, Strings

```csharp
// Value types
int number = 42;
bool isActive = true;
char letter = 'A';

// Reference types
string name = "John";
int[] numbers = {1, 2, 3};
object obj = new object();
```

Q2. What is boxing and unboxing in C#?

Answer: Boxing converts a value type to a reference type, while unboxing does the reverse.

```csharp
// Boxing - value type to reference type
int value = 123;
object boxedValue = value; // Boxing occurs

// Unboxing - reference type to value type
int unboxedValue = (int)boxedValue; // Explicit cast required

// Performance consideration
List<object> items = new List<object>();
for (int i = 0; i < 1000; i++)
{
    items.Add(i); // Boxing happens here - performance impact
}
```

Q3. What is the difference between `var` and `dynamic`?

Answer:

- `var`: Compile-time type inference, strongly typed
- `dynamic`: Runtime type resolution, weakly typed

```csharp
// var - compile time
var name = "John"; // Compiler knows this is string
// name = 123; // Compile error

// dynamic - runtime
dynamic value = "John";
value = 123; // No compile error
value = DateTime.Now; // No compile error
```

Object-Oriented Programming

Q4. Explain the four pillars of OOP in C#

Answer:

1. Encapsulation:

```csharp
public class BankAccount
{
    private decimal _balance; // Private field

    public decimal Balance => _balance; // Read-only property

    public void Deposit(decimal amount)
    {
        if (amount > 0)
            _balance += amount;
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= _balance)
        {
            _balance -= amount;
            return true;
        }
        return false;
    }
}
```

2. Inheritance:

```csharp
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Some generic animal sound");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}
```

3. Polymorphism:

```csharp
public abstract class Shape
{
    public abstract double CalculateArea();
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override double CalculateArea()
    {
        return Width * Height;
    }
}

// Usage
Shape[] shapes = { new Circle { Radius = 5 }, new Rectangle { Width = 4, Height = 6 } };
foreach (Shape shape in shapes)
{
    Console.WriteLine($"Area: {shape.CalculateArea()}"); // Polymorphic behavior
}
```

4. Abstraction:

```csharp
public interface IPaymentProcessor
{
    bool ProcessPayment(decimal amount);
    string GetTransactionId();
}

public class CreditCardProcessor : IPaymentProcessor
{
    public bool ProcessPayment(decimal amount)
    {
        // Credit card specific logic
        return true;
    }

    public string GetTransactionId()
    {
        return Guid.NewGuid().ToString();
    }
}
```

Q5. What is the difference between abstract class and interface?

Answer:

Abstract Class	Interface
Can have implementation	Only contracts (C# 8+ allows default implementations)
Can have fields	Cannot have fields
Single inheritance	Multiple inheritance
Can have constructors	Cannot have constructors

```csharp
// Abstract class
public abstract class Vehicle
{
    protected string _brand; // Field allowed

    public Vehicle(string brand) // Constructor allowed
    {
        _brand = brand;
    }

    public void StartEngine() // Implemented method
    {
        Console.WriteLine("Engine started");
    }

    public abstract void Move(); // Abstract method
}

// Interface
public interface IDriveable
{
    void Drive(); // Contract only
    int MaxSpeed { get; set; } // Property contract
}

public class Car : Vehicle, IDriveable
{
    public Car(string brand) : base(brand) { }

    public override void Move()
    {
        Console.WriteLine("Car is moving");
    }

    public void Drive()
    {
        Console.WriteLine("Driving the car");
    }

    public int MaxSpeed { get; set; } = 200;
}
```

Memory Management

Q6. Explain garbage collection in C#

Answer: The Garbage Collector (GC) automatically manages memory by reclaiming memory used by objects that are no longer reachable.

```csharp
public class MemoryExample
{
    public void DemonstrateGC()
    {
        // Creating objects
        for (int i = 0; i < 1000; i++)
        {
            var obj = new LargeObject();
            // obj goes out of scope and becomes eligible for GC
        }

        // Force garbage collection (not recommended in production)
        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine($"Memory before: {GC.GetTotalMemory(false)}");
        Console.WriteLine($"Memory after: {GC.GetTotalMemory(true)}");
    }
}

public class LargeObject
{
    private byte[] _data = new byte[1000];
}
```

Q7. What is the difference between stack and heap memory?

Answer:

Stack:

- Stores value types and method parameters
- LIFO (Last In, First Out)
- Automatic cleanup when scope ends
- Faster access

Heap:

- Stores reference types
- Managed by Garbage Collector
- Slower access
- Shared across threads

```csharp
public void StackHeapExample()
{
    // Stack allocation
    int stackVariable = 42; // Stored on stack

    // Heap allocation
    string heapVariable = "Hello"; // Reference on stack, object on heap
    List<int> list = new List<int>(); // Reference on stack, object on heap

    ProcessData(stackVariable, heapVariable);
} // stackVariable automatically cleaned up

private void ProcessData(int value, string text)
{
    // Parameters are on stack
    int localValue = value * 2; // Stack
    // When method ends, stack frame is cleaned up
}
```

Collections and LINQ

Q8. What are the different types of collections in C#?

Answer:

```csharp
// Generic Collections
List<string> list = new List<string> { "A", "B", "C" };
Dictionary<string, int> dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 2
};
HashSet<int> set = new HashSet<int> { 1, 2, 3, 2 }; // Unique values only
Queue<string> queue = new Queue<string>();
Stack<int> stack = new Stack<int>();

// Concurrent Collections (Thread-safe)
ConcurrentDictionary<string, int> concurrentDict = new ConcurrentDictionary<string, int>()
ConcurrentQueue<string> concurrentQueue = new ConcurrentQueue<string>();
```

Q9. Explain LINQ with examples

Answer: LINQ (Language Integrated Query) provides query capabilities directly in C#.

```csharp
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }
    public DateTime HireDate { get; set; }
}

public void LinqExamples()
{
    List<Employee> employees = new List<Employee>
    {
        new Employee { Id = 1, Name = "John", Department = "IT", Salary = 75000, HireDate
        new Employee { Id = 2, Name = "Jane", Department = "HR", Salary = 65000, HireDate
        new Employee { Id = 3, Name = "Bob", Department = "IT", Salary = 80000, HireDate
    };

    // Method Syntax
    var highEarners = employees
        .Where(e => e.Salary > 70000)
        .OrderBy(e => e.Name)
        .Select(e => new { e.Name, e.Salary })
        .ToList();

    // Query Syntax
    var itEmployees = from emp in employees
                      where emp.Department == "IT"
                      orderby emp.HireDate descending
                      select emp;

    // Aggregation operations
    var avgSalary = employees.Average(e => e.Salary);
    var totalEmployees = employees.Count();
    var maxSalary = employees.Max(e => e.Salary);

    // Grouping
    var employeesByDept = employees
        .GroupBy(e => e.Department)
        .Select(g => new { Department = g.Key, Count = g.Count(), AvgSalary = g.Average(e
}
```

Async/Await and Threading

Q10. Explain async/await pattern in C#

Answer: Async/await enables asynchronous programming, allowing methods to run without blocking the calling thread.

```csharp
public class AsyncExample
{
    // Async method returning Task
    public async Task<string> FetchDataAsync(string url)
    {
        using HttpClient client = new HttpClient();

        try
        {
            // Await suspends the method until the operation completes
            string response = await client.GetStringAsync(url);
            return response;
        }
        catch (HttpRequestException ex)
        {
            Console.WriteLine($"Error fetching data: {ex.Message}");
            return string.Empty;
        }
    }

    // Async method returning Task (void equivalent)
    public async Task ProcessMultipleUrlsAsync(List<string> urls)
    {
        // Sequential processing
        foreach (string url in urls)
        {
            string data = await FetchDataAsync(url);
            Console.WriteLine($"Processed {url}: {data.Length} characters");
        }

        // Parallel processing
        Task<string>[] tasks = urls.Select(url => FetchDataAsync(url)).ToArray();
        string[] results = await Task.WhenAll(tasks);

        for (int i = 0; i < urls.Count; i++)
        {
            Console.WriteLine($"Parallel result {urls[i]}: {results[i].Length} characters'
        }
    }

    // CPU-bound work with Task.Run
    public async Task<int> CalculateAsync(int n)
    {
        // Offload CPU-intensive work to thread pool
        return await Task.Run(() =>
        {
            int result = 0;
            for (int i = 0; i < n; i++)
            {
                result += i;
            }
            return result;
        });
    }
}
```

Q11. What's the difference between Task.Run, Task.Factory.StartNew, and async/await?

Answer:

```csharp
public class TaskComparison
{
    public async Task CompareTaskMethods()
    {
        // Task.Run - preferred for CPU-bound work
        Task<int> task1 = Task.Run(() => PerformCpuIntensiveWork());

        // Task.Factory.StartNew - more control but complex
        Task<int> task2 = Task.Factory.StartNew(
            () => PerformCpuIntensiveWork(),
            CancellationToken.None,
            TaskCreationOptions.DenyChildAttach,
            TaskScheduler.Default);

        // Async/await - for I/O bound operations
        int result3 = await PerformIoOperationAsync();

        int result1 = await task1;
        int result2 = await task2;

        Console.WriteLine($"Results: {result1}, {result2}, {result3}");
    }

    private int PerformCpuIntensiveWork()
    {
        // Simulate CPU work
        Thread.Sleep(1000);
        return 42;
    }

    private async Task<int> PerformIoOperationAsync()
    {
        // Simulate I/O work
        await Task.Delay(1000);
        return 42;
    }
}
```

Q12. Explain ConfigureAwait(false) and its importance

Answer:

```csharp
public class ConfigureAwaitExample
{
    // Library method - should use ConfigureAwait(false)
    public async Task<string> LibraryMethodAsync()
    {
        // Don't capture synchronization context
        await SomeAsyncOperation().ConfigureAwait(false);

        // This continuation runs on thread pool thread
        return "Library result";
    }

    // UI method - can omit ConfigureAwait or use ConfigureAwait(true)
    public async Task UIMethodAsync()
    {
        string result = await LibraryMethodAsync(); // Default captures context

        // This runs on UI thread, safe to update UI
        UpdateUI(result);
    }

    private async Task SomeAsyncOperation()
    {
        await Task.Delay(100);
    }

    private void UpdateUI(string result)
    {
        // UI update code
        Console.WriteLine($"UI Updated: {result}");
    }
}
```

Q13. Threading: Mutex vs Lock vs Semaphore

Answer: Different synchronization primitives for thread coordination:

csharp

```
public class ThreadSynchronization
{
    private readonly object _lockObject = new object();
    private readonly Mutex _mutex = new Mutex();
    private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(2, 2); // Allow 2 concu

    // Lock - fastest, process-local, non-reentrant for different objects
    public void LockExample()
    {
        lock (_lockObject)
        {
            // Only one thread can execute this block at a time
            Console.WriteLine($"Lock: Thread {Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(1000);
        }
    }

    // Mutex - cross-process, slower, can be named
    public void MutexExample()
    {
        try
        {
            _mutex.WaitOne(); // Acquire mutex
            Console.WriteLine($"Mutex: Thread {Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(1000);
        }
        finally
        {
            _mutex.ReleaseMutex(); // Always release in finally
        }
    }

    // Semaphore - allows N threads simultaneously
    public async Task SemaphoreExampleAsync()
    {
        await _semaphore.WaitAsync(); // Acquire semaphore
        try
        {
            Console.WriteLine($"Semaphore: Thread {Thread.CurrentThread.ManagedThreadId}"
            await Task.Delay(1000);
        }
        finally
        {
            _semaphore.Release(); // Release semaphore
        }
    }

    // Demonstration
    public void DemonstrateAll()
    {
        // Test Lock
        Parallel.For(0, 5, i =>
        {
            LockExample();
        });

        // Test Mutex
        Parallel.For(0, 5, i =>
        {
            MutexExample();
        });

        // Test Semaphore
        Task[] semaphoreTasks = Enumerable.Range(0, 5)
            .Select(_ => SemaphoreExampleAsync())
            .ToArray();
```

```
        Task.WaitAll(semaphoreTasks);
    }
}
```

Q14. Producer-Consumer Pattern with WorkItems

Answer:

csharp

```csharp
public class WorkItem
{
    public int Id { get; set; }
    public string Data { get; set; }
}

public class ProducerConsumerExample
{
    private readonly ConcurrentQueue<WorkItem> _workItems = new ConcurrentQueue<WorkItem>(
    private readonly CancellationTokenSource _cancellationTokenSource = new CancellationT

    // Producer
    public async Task ProducerAsync()
    {
        int itemId = 1;

        while (!_cancellationTokenSource.Token.IsCancellationRequested)
        {
            _workItems.Enqueue(new WorkItem { Id = itemId++, Data = $"Work {itemId}" });

            Console.WriteLine($"Produced work item {itemId}");
            await Task.Delay(500); // Simulate work
        }
    }

    // Consumer
    public async Task ConsumerAsync()
    {
        while (!_cancellationTokenSource.Token.IsCancellationRequested)
        {
            if (_workItems.TryDequeue(out WorkItem item))
            {
                await ProcessWorkItemAsync(item);
            }
            else
            {
                await Task.Delay(100); // Wait if no items
            }
        }
    }

    private async Task ProcessWorkItemAsync(WorkItem item)
    {
        Console.WriteLine($"Processing work item {item.Id}: {item.Data}");
        await Task.Delay(1000); // Simulate processing
        Console.WriteLine($"Completed work item {item.Id}");
    }

    // Using Channel for better producer-consumer pattern
    public async Task ChannelBasedExample()
    {
        var channel = Channel.CreateUnbounded<WorkItem>();
        var writer = channel.Writer;
        var reader = channel.Reader;

        // Producer task
        var producerTask = Task.Run(async () =>
        {
            for (int i = 1; i <= 10; i++)
            {
                await writer.WriteAsync(new WorkItem { Id = i, Data = $"Work {i}" });
                await Task.Delay(100);
            }
            writer.Complete();
        });
```

```csharp
        // Consumer task
        var consumerTask = Task.Run(async () =>
        {
            await foreach (var item in reader.ReadAllAsync())
            {
                await ProcessWorkItemAsync(item);
            }
        });

        await Task.WhenAll(producerTask, consumerTask);
    }

    public async Task RunExample()
    {
        // Start producer and consumer
        var producerTask = ProducerAsync();
        var consumerTask = ConsumerAsync();

        // Let them run for 5 seconds
        await Task.Delay(5000);

        // Stop the operation
        _cancellationTokenSource.Cancel();

        try
        {
            await Task.WhenAll(producerTask, consumerTask);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Operation cancelled");
        }

        // Complete remaining work items
        while (_workItems.TryDequeue(out WorkItem item))
        {
            Console.WriteLine($"Completing remaining item: {item.Id}");
        }
    }
}
```

Exception Handling

Q15. Best practices for exception handling in C#

Answer:

csharp

```
public class ExceptionHandlingBestPractices
{
    // Specific exceptions first, general exceptions last
    public async Task<string> ReadFileAsync(string filePath)
    {
        try
        {
            return await File.ReadAllTextAsync(filePath);
        }
        catch (FileNotFoundException ex)
        {
            // Handle specific exception
            Console.WriteLine($"File not found: {ex.FileName}");
            return string.Empty;
        }
        catch (UnauthorizedAccessException ex)
        {
            // Handle specific exception
            Console.WriteLine($"Access denied: {ex.Message}");
            throw; // Re-throw if caller should handle
        }
        catch (IOException ex)
        {
            // Handle I/O exceptions
            Console.WriteLine($"I/O error: {ex.Message}");
            return string.Empty;
        }
        catch (Exception ex)
        {
            // Log unexpected exceptions
            Console.WriteLine($"Unexpected error: {ex}");
            throw; // Don't swallow unexpected exceptions
        }
    }

    // Custom exceptions
    public class BusinessLogicException : Exception
    {
        public string ErrorCode { get; }

        public BusinessLogicException(string errorCode, string message)
            : base(message)
        {
            ErrorCode = errorCode;
        }

        public BusinessLogicException(string errorCode, string message, Exception innerEx
            : base(message, innerException)
        {
            ErrorCode = errorCode;
        }
    }

    // Using custom exceptions
    public decimal CalculateInterest(decimal principal, decimal rate)
    {
        if (principal < 0)
            throw new BusinessLogicException("INVALID_PRINCIPAL", "Principal amount cannot

        if (rate < 0 || rate > 1)
            throw new BusinessLogicException("INVALID_RATE", "Interest rate must be betwee

        return principal * rate;
    }

    // Exception handling in async methods
```

```
    public async Task<List<string>> ProcessMultipleFilesAsync(string[] filePaths)
    {
        var results = new List<string>();
        var exceptions = new List<Exception>();

        foreach (string filePath in filePaths)
        {
            try
            {
                string content = await ReadFileAsync(filePath);
                results.Add(content);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
                // Continue processing other files
            }
        }

        if (exceptions.Any())
        {
            throw new AggregateException("Multiple files failed to process", exceptions);
        }

        return results;
    }
}
```

Advanced Topics

Q16. What are delegates and events in C#?

Answer:

csharp

```csharp
// Delegate declaration
public delegate void NotificationHandler(string message);
public delegate TResult Func<T, TResult>(T input);

public class EventPublisher
{
    // Event based on delegate
    public event NotificationHandler OnNotification;

    // Generic event with EventArgs
    public event EventHandler<MessageEventArgs> OnMessage;

    protected virtual void RaiseNotification(string message)
    {
        OnNotification?.Invoke(message); // Null-conditional operator
    }

    protected virtual void RaiseMessage(string message)
    {
        OnMessage?.Invoke(this, new MessageEventArgs(message));
    }

    public void PublishNotification(string message)
    {
        Console.WriteLine($"Publishing: {message}");
        RaiseNotification(message);
        RaiseMessage(message);
    }
}

public class MessageEventArgs : EventArgs
{
    public string Message { get; }
    public DateTime Timestamp { get; }

    public MessageEventArgs(string message)
    {
        Message = message;
        Timestamp = DateTime.Now;
    }
}

public class EventSubscriber
{
    public void Subscribe(EventPublisher publisher)
    {
        // Subscribe to events
        publisher.OnNotification += HandleNotification;
        publisher.OnMessage += HandleMessage;

        // Using lambda expressions
        publisher.OnNotification += (msg) => Console.WriteLine($"Lambda: {msg}");
    }

    private void HandleNotification(string message)
    {
        Console.WriteLine($"Notification received: {message}");
    }

    private void HandleMessage(object sender, MessageEventArgs e)
    {
        Console.WriteLine($"Message received at {e.Timestamp}: {e.Message}");
    }
}

// Action and Func delegates
```

```csharp
public class DelegateExamples
{
    public void DemonstrateBuiltInDelegates()
    {
        // Action - void return type
        Action<string> printMessage = msg => Console.WriteLine(msg);
        printMessage("Hello World");

        // Func - has return type
        Func<int, int, int> add = (x, y) => x + y;
        int result = add(5, 3);

        // Predicate - returns bool
        Predicate<int> isEven = x => x % 2 == 0;
        bool even = isEven(4);

        // Multicast delegate
        Action combined = () => Console.WriteLine("First");
        combined += () => Console.WriteLine("Second");
        combined += () => Console.WriteLine("Third");
        combined(); // Executes all three
    }
}
```

Q17. What are generics and constraints in C#?

Answer:

csharp

```csharp
// Generic class with constraints
public class Repository<T> where T : class, IEntity, new()
{
    private readonly List<T> _items = new List<T>();

    public void Add(T item)
    {
        if (item == null)
            throw new ArgumentNullException(nameof(item));

        _items.Add(item);
    }

    public T GetById(int id)
    {
        return _items.FirstOrDefault(item => item.Id == id);
    }

    public T CreateNew()
    {
        return new T(); // new() constraint allows this
    }
}

public interface IEntity
{
    int Id { get; set; }
}

public class User : IEntity
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}

// Generic methods
public class GenericMethods
{
    // Method with type constraints
    public T ProcessEntity<T>(T entity) where T : IEntity
    {
        Console.WriteLine($"Processing entity with ID: {entity.Id}");
        return entity;
    }

    // Multiple type parameters
    public TResult Transform<TSource, TResult>(TSource source, Func<TSource, TResult> trai
    {
        return transformer(source);
    }

    // Constraint examples
    public void ConstraintExamples<T>(T item)
        where T : class, IComparable<T>, new()
    {
        // class: reference type
        // IComparable<T>: must implement interface
        // new(): must have parameterless constructor

        T newItem = new T();
        int comparison = item.CompareTo(newItem);
    }
}

// Covariance and Contravariance
```

```csharp
public interface IProducer<out T> // Covariant
{
    T Produce();
}

public interface IConsumer<in T> // Contravariant
{
    void Consume(T item);
}

public class VarianceExample
{
    public void DemonstrateVariance()
    {
        // Covariance - can assign derived to base
        IProducer<string> stringProducer = new StringProducer();
        IProducer<object> objectProducer = stringProducer; // Valid due to covariance

        // Contravariance - can assign base to derived
        IConsumer<object> objectConsumer = new ObjectConsumer();
        IConsumer<string> stringConsumer = objectConsumer; // Valid due to contravariance
    }
}

public class StringProducer : IProducer<string>
{
    public string Produce() => "Hello";
}

public class ObjectConsumer : IConsumer<object>
{
    public void Consume(object item) => Console.WriteLine(item);
}
```

Interview Questions Summary

Common Coding Challenges

1. **Reverse a string without using built-in methods**
2. **Find duplicate elements in an array**
3. **Implement a simple cache with expiration**
4. **Design a thread-safe singleton**
5. **Calculate factorial using recursion and iteration**
6. **Implement async file processing with error handling**

System Design Questions

1. **Design a logging framework**
2. **Implement a connection pool**
3. **Design a simple pub-sub system**
4. **Create a rate limiter**
5. **Design a simple ORM mapping**

Best Practices to Remember

1. **Always dispose of unmanaged resources using `using` statements**
2. **Use `ConfigureAwait(false)` in library code**
3. **Prefer composition over inheritance**
4. **Use specific exception types**
5. **Follow SOLID principles**
6. **Use async/await for I/O operations, Task.Run for CPU-bound work**

7. **Avoid blocking async operations with `.Result` or `.Wait()`**
8. **Use concurrent collections for thread-safe operations**

This guide covers the most important C# concepts you'll encounter in technical interviews. Practice implementing these examples and understanding the underlying principles to succeed in your interviews.