# Types of Classes in C#

C# provides different types of classes, each serving a specific purpose in object-oriented programming. Below are the main types of classes in C#:

## 1. Abstract Class

* **Definition**: An abstract class is a class that cannot be instantiated and is meant to be inherited by other classes.
* **Use**: Used to define a common base structure for multiple derived classes.
* **Advantages**:
  + Encourages code reusability and modularity.
  + Ensures a consistent interface for derived classes.
* **Disadvantages**:
  + Cannot be instantiated directly.
  + All derived classes must implement abstract methods, increasing code complexity.

### Example:

abstract class Animal  
{  
 public abstract void MakeSound();  
 public void Sleep()  
 {  
 Console.WriteLine("Sleeping...");  
 }  
}  
  
class Dog : Animal  
{  
 public override void MakeSound()  
 {  
 Console.WriteLine("Bark");  
 }  
}

## 2. Sealed Class

* **Definition**: A sealed class is a class that cannot be inherited by other classes.
* **Use**: Used when you want to restrict the inheritance of a class.
* **Advantages**:
  + Prevents unwanted modifications through inheritance.
  + Enhances security and stability of the code.
* **Disadvantages**:
  + Cannot be extended for further customization.

### Example:

sealed class MathOperations  
{  
 public int Add(int a, int b)  
 {  
 return a + b;  
 }  
}

## 3. Static Class

* **Definition**: A static class is a class that cannot be instantiated and contains only static members.
* **Use**: Used for utility functions and helper methods.
* **Advantages**:
  + Provides global access to utility functions.
  + Prevents unnecessary instantiations.
* **Disadvantages**:
  + Cannot be inherited or extended.
  + Limited to static members only.

### Example:

static class Utility  
{  
 public static void PrintMessage(string message)  
 {  
 Console.WriteLine(message);  
 }  
}

## 4. Partial Class

* **Definition**: A partial class allows a class definition to be split across multiple files.
* **Use**: Used in large projects where multiple developers work on different parts of the same class.
* **Advantages**:
  + Improves code organization.
  + Allows parallel development by multiple team members.
* **Disadvantages**:
  + Can make debugging and maintenance complex if not managed properly.

### Example:

**File1.cs:**

public partial class Person  
{  
 public string Name { get; set; }  
}

**File2.cs:**

public partial class Person  
{  
 public void Greet()  
 {  
 Console.WriteLine("Hello " + Name);  
 }  
}

## 5. Generic Class

* **Definition**: A generic class allows defining a class with a placeholder for the data type.
* **Use**: Used to create reusable, type-safe code that works with any data type.
* **Advantages**:
  + Enables type safety while maintaining flexibility.
  + Reduces code duplication by allowing the same class to handle different data types.
* **Disadvantages**:
  + Can be complex to implement and understand for beginners.

### Example:

class Box<T>  
{  
 public T Value { get; set; }  
 public void ShowValue()  
 {  
 Console.WriteLine("Value: " + Value);  
 }  
}

## 6. Nested Class

* **Definition**: A nested class is a class defined within another class.
* **Use**: Used for logically grouping classes that are only used within the enclosing class.
* **Advantages**:
  + Helps in encapsulating related functionality.
  + Can access private members of the outer class.
* **Disadvantages**:
  + Increases complexity of the code structure.

### Example:

class OuterClass  
{  
 public class InnerClass  
 {  
 public void Display()  
 {  
 Console.WriteLine("This is a nested class");  
 }  
 }  
}

# Generics in C#

## Introduction

* Generics allow defining **classes, methods, interfaces, and delegates** with type parameters.
* They enable code to be **reusable, type-safe, and optimized for performance**.
* Introduced in **C# 2.0**, generics help avoid redundant code and improve maintainability.

## Key Benefits of Generics

### 1. Code Reusability

* Eliminates the need to write multiple versions of the same class or method for different data types.
* A single generic class or method can work with multiple types.

### 2. Type Safety

* Ensures type correctness **at compile time**, preventing runtime errors caused by invalid type conversions.
* Reduces the need for explicit type casting.

### 3. Performance Optimization

* Avoids **boxing and unboxing** when working with value types, reducing unnecessary memory overhead.
* Enhances execution speed compared to non-generic collections like ArrayList or Hashtable.

### 4. Maintainability and Readability

* Generic code is **easier to maintain** because changes apply to multiple types without duplication.
* Improves readability by removing redundant type definitions.

### 5. Supports Generic Collections

* Generic collections like List<T>, Dictionary<TKey, TValue>, and Queue<T> provide better performance and type safety compared to non-generic collections.

## Usage of Generics in C#

### 1. Generic Class

* A class that can operate on any data type using a placeholder type.

**Example:**

public class GenericContainer<T>  
{  
 private T item;  
 public void SetItem(T value)  
 {  
 item = value;  
 }  
 public T GetItem()  
 {  
 return item;  
 }  
}

### 2. Generic Method

* A method that can work with multiple data types without explicit type definitions.

**Example:**

public class Utility  
{  
 public static void Print<T>(T data)  
 {  
 Console.WriteLine(data);  
 }  
}

### 3. Generic Interface

* Interfaces can be defined with type parameters to create flexible implementations.

**Example:**

public interface IContainer<T>  
{  
 void Add(T item);  
 T Get();  
}

### 4. Generic Constraints

* Constraints restrict the types that can be used with generics.

**Example:**

public class Sample<T> where T : new()  
{  
 public T CreateInstance()  
 {  
 return new T();  
 }  
}

### 5. Generic Delegates

* Delegates can be defined with generics to create flexible function pointers.

**Example:**

public delegate T Operation<T>(T a, T b);

### 6. Generic Collections

* Generic collections in **System.Collections.Generic** provide improved performance and type safety.

**Common Generic Collections:**

| **Collection** | **Description** |
| --- | --- |
| List<T> | Dynamic array that stores elements of type T. |
| Dictionary<TKey, TValue> | Key-value pair collection. |
| Queue<T> | FIFO (First-In-First-Out) collection. |
| Stack<T> | LIFO (Last-In-First-Out) collection. |
| HashSet<T> | Unordered collection of unique elements. |

**Example:**

List<int> numbers = new List<int> { 1, 2, 3, 4 };  
numbers.Add(5);  
Console.WriteLine(numbers[2]); // Output: 3

## Conclusion

* Generics improve **code reusability, type safety, and performance**.
* They are widely used in **collections, interfaces, methods, and delegates**.
* Using generics **reduces redundant code** and ensures compile-time type checking.
* Generic constraints help **limit the allowed types** for better control.

By leveraging generics effectively, developers can write **scalable, efficient, and maintainable** C# applications.

# Data Serialization (JSON, XML)

## Introduction

* Data serialization converts an object into a format for storage or transmission.
* Common formats: **JSON (JavaScript Object Notation)** and **XML (eXtensible Markup Language)**.

## JSON Serialization and Deserialization

### Using Json.NET (Newtonsoft.Json)

* Json.NET is a popular library in .NET for JSON handling.

#### **Serializing an Object to JSON**

using Newtonsoft.Json;  
  
User user1 = new User { Id = 1, Name = "John Doe", Email = "john.doe@example.com" };  
string jsonNetString = JsonConvert.SerializeObject(user1);  
Console.WriteLine(jsonNetString);

#### **Deserializing JSON to an Object**

User deserializedUser = JsonConvert.DeserializeObject<User>(jsonNetString);  
Console.WriteLine(deserializedUser.Name);

### Using System.Text.Json

* .NET Core and .NET 5+ provide System.Text.Json for JSON serialization.

#### **Serializing with System.Text.Json**

using System.Text.Json;  
  
string jsonString = JsonSerializer.Serialize(user1);  
Console.WriteLine(jsonString);

#### **Deserializing with System.Text.Json**

User user2 = JsonSerializer.Deserialize<User>(jsonString);  
Console.WriteLine(user2.Name);

## XML Serialization and Deserialization

* XML serialization converts an object into an XML format.

### Using XmlSerializer

#### **Serializing an Object to XML**

using System.Xml.Serialization;  
using System.IO;  
  
XmlSerializer xmlSerializer = new XmlSerializer(typeof(User));  
using (StringWriter stringWriter = new StringWriter())  
{  
 xmlSerializer.Serialize(stringWriter, user1);  
 string xmlString = stringWriter.ToString();  
 Console.WriteLine(xmlString);  
}

#### **Deserializing XML to an Object**

using (StringReader stringReader = new StringReader(xmlString))  
{  
 User user3 = (User)xmlSerializer.Deserialize(stringReader);  
 Console.WriteLine(user3.Name);  
}

## Using MemoryStream for Serialization

* MemoryStream is used for in-memory serialization and deserialization.

#### **Serializing to a MemoryStream**

MemoryStream memoryStream = new MemoryStream();  
using (StreamWriter writer = new StreamWriter(memoryStream))  
{  
 writer.Write(jsonNetString);  
 writer.Flush();  
 memoryStream.Position = 0; // Reset stream position  
}

#### **Reading from a MemoryStream**

using (StreamReader reader = new StreamReader(memoryStream))  
{  
 string jsonFromStream = reader.ReadToEnd();  
 Console.WriteLine(jsonFromStream);  
}

## Summary

### Comparison Table

| **Feature** | **JSON (Json.NET)** | **JSON (System.Text.Json)** | **XML** |
| --- | --- | --- | --- |
| Readability | High | High | Moderate |
| Performance | Fast | Faster than Json.NET | Slower |
| .NET Built-in Support | No | Yes | Yes |
| Flexibility | High | Moderate | Moderate |

**Use JSON for lightweight and fast data exchange, while XML is useful for structured and hierarchical data representation.**

# 

# Modifying Built-in Libraries in C#

* **Built-in libraries** provide core functionalities to simplify development.
* **Modifications** help meet specific project needs, enhance performance, or customize functionality.

## Ways to Modify Built-in Libraries

* **Extension methods** – Add new methods without modifying the source code.
* **Inheritance** – Create a subclass and override methods.
* **Method overriding** – Modify behavior of virtual or abstract methods.
* **Reflection** – Access and modify private members dynamically at runtime.
* **Source code modification** – Directly modify .NET source code (not recommended).

## Advantages of Modifying Libraries

* **Customization** – Tailor functionalities to specific business needs.
* **Performance optimization** – Improve efficiency and eliminate unnecessary processing.
* **Better integration** – Align library behavior with internal application logic.
* **Security enhancements** – Apply security patches before official updates.
* **Improved debugging** – Enable better error handling and logging.

## Best Practices

* **Use extension methods** instead of modifying original code.
* **Follow open-closed principle** – Extend without altering existing functionality.
* **Document changes** for future reference and maintainability.
* **Ensure compatibility** with future .NET updates.
* **Use reflection carefully** to avoid performance issues.
* **Prefer composition** over inheritance when modifying behavior.

## Conclusion

* Modifying built-in libraries in C# offers flexibility, performance improvements, and security benefits.
* Changes should follow best practices to maintain stability and compatibility.

# 

# Lambda Expressions

## Definition

* Lambda expressions are a concise way to represent anonymous functions (functions without a name).
* They provide a clear and concise syntax for defining small functions inline, reducing boilerplate code.
* In many programming languages, lambda expressions are first-class citizens, meaning they can be:
  + Assigned to variables.
  + Passed as arguments.
  + Returned from functions.

### General Syntax (C# Example):

(parameters) => expression;

Or, for multiple statements:

(parameters) => { statement1; statement2; return value; };

## Uses of Lambda Expressions

1. **Functional Programming**: Used in functional paradigms to pass behavior as arguments.
2. **Event Handling**: Used in UI development and event-driven programming.
3. **LINQ Queries**: Extensively used in C# LINQ for filtering, mapping, and reducing collections.
4. **Concurrency & Parallelism**: Used in parallel computing and multi-threading.
5. **Shortening Code**: Reduces the need for defining separate named functions.
6. **Higher-Order Functions**: Passed as parameters to other functions for custom behavior.
7. **Sorting & Filtering Data**: Often used in collections for sorting and filtering operations.

## Key Features

* **Anonymous**: No need to explicitly define function names.
* **Concise**: Reduces boilerplate code.
* **Higher-Order Functions**: Can be passed as arguments or returned.
* **Lexical Scoping**: Can access variables in its scope.

## Additional Information

* **Closures**: Lambda expressions can capture variables from their enclosing scope.
* **Performance Considerations**: Overuse of lambda expressions in performance-critical applications can sometimes lead to reduced readability and potential memory leaks if references are not managed properly.
* **Comparison with Named Functions**: While lambda expressions improve code brevity, they may reduce readability if overused.
* **Limitations in Some Languages**: Some languages restrict lambda expressions from containing complex statements or require explicit return types.

# Extension Methods in C#

## Overview

* Extension methods in C# allow adding new functionality to existing types (including built-in .NET classes) without modifying their source code.
* They enable developers to create reusable and customized methods.
* Enhance readability and improve functionality.

## Key Features

* Extend functionality of existing classes without modification.
* Improve code readability and maintainability.
* Work with built-in .NET types and user-defined types.
* Defined as static methods within static classes.
* First parameter must use the this keyword to specify the type being extended.

## Syntax and Implementation

To define an extension method: - Create a static class. - Define a static method inside it. - Use this before the first parameter to specify the type to be extended.

**Example:**

using System;  
  
public static class StringExtensions  
{  
 public static bool IsNullOrEmpty(this string str)  
 {  
 return string.IsNullOrEmpty(str);  
 }  
}  
  
class Program  
{  
 static void Main()  
 {  
 string text = "Hello";  
 Console.WriteLine(text.IsNullOrEmpty()); // Output: False  
 }  
}

## Benefits of Using Extension Methods

* **Enhances Readability:** Methods can be called as if they are part of the extended type.
* **Improves Code Reusability:** Custom functionalities can be reused across multiple projects.
* **Non-Intrusive:** Does not require modification of original class definitions.
* **Supports Built-in and Custom Classes:** Works seamlessly with .NET framework types and user-defined classes.

## Limitations

* Cannot override existing instance methods.
* May lead to confusion if overused, as they do not appear in the original class definition.
* Performance impact is minimal but should be used wisely.

## Best Practices

* Use extension methods for utility functions that are widely used.
* Avoid overusing them, especially when subclassing is a better option.
* Follow naming conventions to maintain clarity.
* Document properly to ensure understanding and maintainability.

# 

# LINQ (Language Integrated Query) - Documentation

## 1. Definition of LINQ

LINQ (Language Integrated Query) is a powerful feature in C# that provides a consistent query syntax to retrieve and manipulate data from different sources like collections, databases, XML, and more. It enables querying using a uniform syntax irrespective of the underlying data source.

## 2. Uses of LINQ

* Simplifies data querying in C# with an intuitive, SQL-like syntax.
* Improves code readability and maintainability.
* Reduces the need for complex loops and conditional logic.
* Provides a single, consistent query model for various data sources.

## 3. Different Ways to Apply LINQ

LINQ can be applied in two ways:

### a. Query Syntax

Query syntax is similar to SQL and is easier to read for those familiar with database queries.

var result = from student in students  
 where student.Age > 20  
 orderby student.Name  
 select student;

### b. Method Syntax

Method syntax uses extension methods and lambda expressions, making it more concise.

var result = students.Where(student => student.Age > 20)  
 .OrderBy(student => student.Name)  
 .ToList();

## LINQ can be applied to various data sources, including:

### LINQ on Lists (Collections)

LINQ can be used to filter, sort, and aggregate data from generic collections such as List<T>.

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();

### LINQ on DataTable (Using AsEnumerable)

LINQ can be used on DataTable by converting it to IEnumerable<T> using AsEnumerable().

DataTable dt = new DataTable();  
dt.Columns.Add("Id", typeof(int));  
dt.Columns.Add("Name", typeof(string));  
dt.Rows.Add(1, "Alice");  
dt.Rows.Add(2, "Bob");  
var filteredRows = dt.AsEnumerable().Where(row => row.Field<int>("Id") > 1);

### LINQ on Files

LINQ can be used to process file data, such as reading lines from a text file.

var lines = File.ReadAllLines("sample.txt");  
var filteredLines = lines.Where(line => line.Contains("error"));

### LINQ on Objects

LINQ can be applied to object collections to filter or transform data.

class Student {  
 public string Name { get; set; }  
 public int Age { get; set; }  
}  
  
List<Student> students = new List<Student> {  
 new Student { Name = "John", Age = 22 },  
 new Student { Name = "Sara", Age = 24 }  
};  
  
var youngStudents = students.Where(s => s.Age < 23);

## 4. List of LINQ Keywords and Their Use

### Basics

| **Keyword** | **Use** |
| --- | --- |
| from | Declares the data source. |
| where | Filters the data based on conditions. |
| select | Projects (selects) specific data fields. |
| orderby | Sorts data in ascending/descending order. |
| group by | Groups data based on a key. |
| join | Combines multiple data sources. |
| into | Stores results into a temporary variable. |
| let | Defines a variable in a query. |

### Filtering and Searching

| **Keyword** | **Use** |
| --- | --- |
| distinct | Removes duplicate values. |
| first | Returns the first matching element. |
| firstOrDefault | Returns the first matching element or default value if none found. |
| single | Returns a single matching element. |
| singleOrDefault | Returns a single matching element or default value if none found. |
| any | Checks if any elements satisfy the condition. |
| all | Checks if all elements satisfy the condition. |

### Aggregates and Calculations

| **Keyword** | **Use** |
| --- | --- |
| count | Returns the number of elements in a collection. |
| sum | Calculates the sum of numeric elements. |
| average | Calculates the average of numeric elements. |

### Ordering

| **Keyword** | **Use** |
| --- | --- |
| orderby | Sorts elements in ascending order. |
| orderby descending | Sorts elements in descending order. |
| thenby | Performs a secondary sorting in ascending order. |
| thenby descending | Performs a secondary sorting in descending order. |

### Grouping

| **Keyword** | **Use** |
| --- | --- |
| group by | Groups elements based on a specified key. |

## 5. Different Types of Joins in LINQ

### a. Inner Join

var result = from s in students  
 join c in courses on s.CourseId equals c.Id  
 select new { s.Name, c.CourseName };

### b. Left Join

var result = from s in students  
 join c in courses on s.CourseId equals c.Id into studentCourses  
 from sc in studentCourses.DefaultIfEmpty()  
 select new { s.Name, CourseName = sc?.CourseName ?? "No Course" };

### c. Group Join

var result = from c in courses  
 join s in students on c.Id equals s.CourseId into studentGroup  
 select new { CourseName = c.CourseName, Students = studentGroup };

## 6. Using GroupBy in LINQ

var groupedStudents = students.GroupBy(s => s.Age)  
 .Select(group => new { Age = group.Key, Students = group.ToList() });

## 7. LINQ to Array Operations

LINQ can be used on arrays for filtering, sorting, and aggregating data.

int[] numbers = { 3, 7, 1, 9, 4 };  
var sortedNumbers = numbers.OrderBy(n => n).ToArray();

### Common Operations on Arrays

| **Operation** | **Example** |
| --- | --- |
| Filtering | var evenNumbers = numbers.Where(n => n % 2 == 0).ToArray(); |
| Sorting | var sortedNumbers = numbers.OrderBy(n => n).ToArray(); |
| Aggregation | var sum = numbers.Sum(); |
| Searching | var firstEven = numbers.First(n => n % 2 == 0); |

This documentation provides an overview of LINQ, its applications, query and method syntax, keywords, joins, groupBy, and array operations. LINQ is a versatile and powerful tool for querying and transforming data efficiently in C#.

# 

# Service Stack OrmLite with MySQL - Documentation

## Introduction

ServiceStack OrmLite is a lightweight, fast, and simple ORM for .NET that provides strong-typed SQL operations with minimal configuration. This document outlines the steps required to integrate ServiceStack OrmLite with MySQL, including installation, configuration, and basic CRUD operations.

## Installation

To use OrmLite with MySQL in your .NET Core application, install the required NuGet package:

Install-Package ServiceStack.OrmLite.MySql

For MySQL 8+, install:

Install-Package ServiceStack.OrmLite.MySqlConnector

## Setting Up MySQL Connection

To establish a database connection, create a DbConnectionFactory using MySQL connection settings.

### Example - Setting Up Connection

using ServiceStack.OrmLite;  
using ServiceStack.OrmLite.MySql;  
using System.Data;  
  
public class DBConnection  
{  
 private readonly string \_connectionString;  
 private readonly IDbConnectionFactory \_dbFactory;  
  
 public DBConnection(string connectionString)  
 {  
 \_connectionString = connectionString;  
 \_dbFactory = new OrmLiteConnectionFactory(\_connectionString, MySqlDialect.Provider);  
 }  
  
 public IDbConnection OpenConnection() => \_dbFactory.OpenDbConnection();  
}

**Example Connection String:**

string connectionString = "Server=localhost;Database=yourdb;User=root;Password=yourpassword;SslMode=none;";

## Create Tables:

public class User  
{  
 [AutoIncrement]  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public string Email { get; set; }  
}

### Initialize Database Tables

using (var db = dbFactory.OpenDbConnection())  
{  
 db.CreateTableIfNotExists<User>();  
}

## CRUD Operations

### Insert Data

var newUser = new User { Name = "John Doe", Email = "john@example.com" };  
db.Insert(newUser);

### Fetch Data

List<User> users = db.Select<User>();

### Update Data

db.Update<User>(new { Name = "Jane Doe" }, where: x => x.Id == 1);

### Delete Data

db.Delete<User>(x => x.Id == 1);

## Select Queries

| **Method** | **Description** |
| --- | --- |
| db.Select<T>() | Retrieves all records of type T |
| db.Single<T>(id) | Retrieves a single record by primary key |
| db.Single<T>(expression) | Retrieves the first record matching a condition |
| db.Select<T>(sqlExpression) | Retrieves records based on a SQL expression |
| db.Select<T>(q => q.Field == value) | Retrieves records based on LINQ-style condition |
| db.Exists<T>(q => q.Field == value) | Checks if a record exists matching the condition |
| db.Count<T>() | Gets the total number of records in the table |
| db.Count<T>(q => q.Field == value) | Counts records matching a condition |
| db.Select<T>(limit) | Retrieves a limited number of records |
| db.Select<T>(orderby: q => q.Field) | Retrieves sorted records based on a field |
| db.Select<T>(q => q.Field.Contains(“value”)) | Retrieves records where a field contains a substring |
| db.Select<T>(q => q.Field.StartsWith(“value”)) | Retrieves records where a field starts with a value |
| db.Select<T>(q => q.Field.EndsWith(“value”)) | Retrieves records where a field ends with a value |
| db.SingleById<T>(id) | Retrieves a single record by its ID |
| db.From<T>() | Creates an SqlExpression<T> for query building |
| db.Scalar<T>(expression) | Retrieves a scalar value based on the expression |
| db.Column<T>(expression) | Retrieves a single column from the table |
| db.ColumnDistinct<T>(expression) | Retrieves distinct values of a column |
| db.Lookup<K, V>(expression) | Retrieves a lookup dictionary mapping keys to values |
| db.Dictionary<K, V>(expression) | Retrieves a dictionary of key-value pairs |
| db.Select<T>(q => Sql.In(q.Field, new[]{1,2,3})) | Retrieves records where the field value is in a given set |
| db.Join<T1, T2>(expression) | Performs a SQL JOIN between two tables |

Example Usage:

var users = db.Select<User>(q => q.Name == "John Doe");  
var firstUser = db.Single<User>(q => q.Email == "john@example.com");  
var topUsers = db.Select<User>(limit: 5);  
var userById = db.SingleById<User>(1);  
var emails = db.Column<string>(db.From<User>().Select(x => x.Email));  
var exists = db.Exists<User>(q => q.Email == "john@example.com");

## Transactions

To perform atomic operations:

using (var db = dbFactory.OpenDbConnection())  
using (var transaction = db.OpenTransaction())  
{  
 db.Insert(new User { Name = "Alice", Email = "alice@example.com" });  
 db.Insert(new User { Name = "Bob", Email = "bob@example.com" });  
 transaction.Commit();  
}

# Security & Cryptography: AES vs. Rijndael

## Introduction

**Definitions of AES and Rijndael:**

* **AES (Advanced Encryption Standard):** A symmetric encryption algorithm standardized by NIST, derived from the Rijndael cipher, with fixed block size (128 bits) and key sizes of 128, 192, or 256 bits.
* **Rijndael Cipher:** A flexible encryption algorithm developed by Vincent Rijmen and Joan Daemen, supporting variable block sizes (128, 160, 192, 224, or 256 bits) and key sizes of 128, 192, or 256 bits.

## Difference Between AES and Rijndael

AES and Rijndael are closely related but have some key differences:

| **Feature** | **AES** | **Rijndael** |
| --- | --- | --- |
| Key Sizes | 128, 192, 256 bits | 128, 192, 256 bits |
| Block Sizes | Fixed at 128 bits | Variable: 128, 160, 192, 224, 256 bits |
| Rounds | 10, 12, 14 (based on key size) | Varies depending on block and key sizes |
| Standardized By | NIST (as AES) | Developed by Vincent Rijmen and Joan Daemen |
| Flexibility | Standardized and widely used | More flexible but less commonly implemented |

## Implementation Steps

### 1. Implementing AES

AES can be implemented using various programming languages and libraries. Below is a general approach:

#### **Step 1: Key Generation**

* Choose a key size: 128, 192, or 256 bits.
* Generate a random key securely using a cryptographic library.

#### **Step 2: Encryption Process**

1. Convert plaintext to a byte array.
2. Generate an Initialization Vector (IV) for CBC mode (if required).
3. Apply AES encryption using a selected mode (e.g., ECB, CBC, GCM).
4. Obtain the ciphertext.

#### **Step 3: Decryption Process**

1. Retrieve the IV (if used).
2. Apply AES decryption with the same key.
3. Convert the decrypted bytes back to readable text.

### 2. Implementing Rijndael

Rijndael supports flexible block sizes, so implementation steps are similar to AES but with additional flexibility.

#### **Step 1: Key and Block Size Selection**

* Choose a key size: 128, 192, 256 bits.
* Choose a block size: 128, 160, 192, 224, or 256 bits.

#### **Step 2: Encryption Process**

1. Convert plaintext to a byte array.
2. Apply

**Database with C# - Basics and Implementation Guide**

**1. Introduction**

A database is an organized collection of data that can be accessed, managed, and updated efficiently. In C#, databases are commonly used to store and manipulate structured data, typically through SQL-based relational databases like:

* MySQL
* SQL Server
* PostgreSQL

**2. Purpose of Using a Database with C#**

* **Persistent Data Storage**: Stores data persistently across application restarts.
* **Efficient Data Management**: Enables structured querying and filtering of data.
* **Scalability**: Handles large amounts of data efficiently.
* **Data Integrity and Security**: Ensures data consistency and access control.

**3. When to Use a Manually Managed Database**

A manually managed database (directly interacting with ADO.NET or Entity Framework Core without an ORM like OrmLite) is preferred when:

* **Custom Query Optimization is Required**: Allows precise control over queries and indexing.
* **High Performance is a Priority**: Eliminates ORM overhead for speed-critical applications.
* **Complex Transactions and Relationships**: When advanced transactions, joins, and stored procedures are used.
* **Fine-grained Connection Management**: Offers better control over database connections and resources.

**4. Benefits of Manual Database Handling Over OrmLite**

* **Greater Control**: Directly execute optimized SQL queries without ORM abstraction.
* **Better Performance**: Reduces overhead caused by ORM frameworks.
* **Custom Transaction Handling**: Allows manual transaction control for complex scenarios.
* **More Flexibility**: Supports database-specific optimizations and features.
* **No Learning Curve**: Avoids the need to learn an additional ORM framework.

**5. Basic Flow to Implement a Database in C#**

**Step 1: Choose and Setup the Database**

1. Install a database server like MySQL, SQL Server, or PostgreSQL.
2. Install the necessary NuGet packages:
   * Using Package Manager Console:
   * Install-Package MySql.Data
   * Using .NET CLI:
   * dotnet add package MySql.Data
3. Create a new database:
4. CREATE DATABASE myDB;
5. Select the database:
6. USE myDB;
7. Define tables, e.g., creating a Users table:
8. CREATE TABLE Users (
9. ID INT PRIMARY KEY AUTO\_INCREMENT,
10. Name VARCHAR(100),
11. Age INT
12. );

**Step 2: Establish a Connection**

Use ADO.NET to establish a connection with the database.

using System;

using System.Data.SqlClient;

class Program

{

static void Main()

{

string connectionString = "Server=myServer;Database=myDB;User Id=myUser;Password=myPass;";

using (SqlConnection conn = new SqlConnection(connectionString))

{

conn.Open();

Console.WriteLine("Database connected successfully!");

}

}

}

**Step 3: Create and Execute SQL Commands**

Use SqlCommand to execute SQL queries.

string query = "INSERT INTO Users (Name, Age) VALUES (@Name, @Age)";

using (SqlCommand cmd = new SqlCommand(query, conn))

{

cmd.Parameters.AddWithValue("@Name", "John");

cmd.Parameters.AddWithValue("@Age", 30);

cmd.ExecuteNonQuery();

}

**Step 4: Read Data from Database**

string selectQuery = "SELECT \* FROM Users";

using (SqlCommand cmd = new SqlCommand(selectQuery, conn))

{

using (SqlDataReader reader = cmd.ExecuteReader())

{

while (reader.Read())

{

Console.WriteLine($"ID: {reader["ID"]}, Name: {reader["Name"]}, Age: {reader["Age"]}");

}

}

}

**Step 5: Update and Delete Data**

**Update Example:**

string updateQuery = "UPDATE Users SET Age = @Age WHERE Name = @Name";

using (SqlCommand cmd = new SqlCommand(updateQuery, conn))

{

cmd.Parameters.AddWithValue("@Name", "John");

cmd.Parameters.AddWithValue("@Age", 31);

cmd.ExecuteNonQuery();

}

**Delete Example:**

string deleteQuery = "DELETE FROM Users WHERE Name = @Name";

using (SqlCommand cmd = new SqlCommand(deleteQuery, conn))

{

cmd.Parameters.AddWithValue("@Name", "John");

cmd.ExecuteNonQuery();

}

**Step 6: Close the Connection**

Connections should be closed automatically when using using blocks. However, explicitly closing can be done using:

conn.Close();