**1. Klassenbibliotheken im .NET Framework**

Das .NET Framework besteht aus zahlreichen (über 470) DLL-Dateien (Dynamic-Link Libraries), die in verschiedenen Bibliotheken vielseitige Funktionalitäten bereitstellen.

Eine typische Installation des .NET Framework umfasst **hunderte von DLLs**. Diese umfassen Bibliotheken für:

* Grundlegende Funktionalitäten
* Webentwicklung (ASP.NET)
* Datenzugriff (ADO.NET, Entity Framework)
* Windows Forms
* U.v.m.

**Beispiele von DLLs im .NET Framework:**

**mscorlib.dll** - Die Basisbibliothek für grundlegende .NET-Funktionen (z. B. Collections, Strings, etc.).

**System.dll** - Stellt grundlegende Namespaces wie System bereit.

**System.Core.dll** - Enthält LINQ und andere Erweiterungen.

**System.Data.dll** - Unterstützt den Datenzugriff, z. B. ADO.NET.

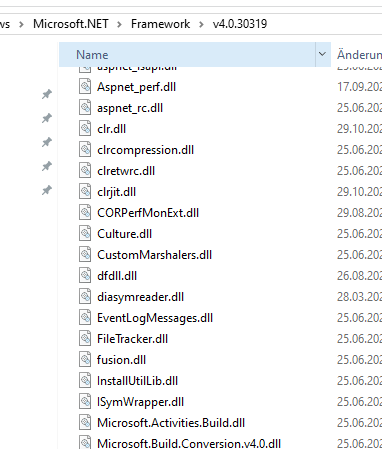
**System.Xml.dll** - Bietet XML-Verarbeitungsfunktionen.

**System.Windows.Forms.dll** - Für die Entwicklung von Windows-Desktop-Anwendungen.

**System.Web.dll** - Unterstützt Webanwendungen und ASP.NET.

**DLLs im .Net-Framework (Ausschnitt)**

C:\Windows\Microsoft.NET\Framework64\v4.0.30319



Wie finde ich heraus, in welcher Assembly (DLL) ein bestimmter Typ sich befindet?

var type = typeof(System.Console);

Console.WriteLine($"Namespace: {type.Namespace}");

Console.WriteLine($"Assembly: {type.Assembly.FullName}");

**2. NuGet-Paketmanager**

Was ist NuGet?

* NuGet ist ein Paketmanager für .NET, mit dem externe Bibliotheken und Tools einfach in Projekte integriert werden können.
* Es ermöglicht die einfache Verwaltung von Abhängigkeiten und Versionen in Projekten.

Warum NuGet verwenden?

* Zeitersparnis: Bibliotheken **müssen nicht manuell heruntergeladen** und eingebunden werden.
* Einfaches **Versionsmanagement**.
* Zentrale Verwaltung und Updates.

Wie funktioniert NuGet?

* NuGet-Pakete enthalten Bibliotheken, Dokumentation und Abhängigkeiten.
* Pakete werden aus öffentlichen oder privaten NuGet-Repositories bezogen.

Beispiele für bekannte NuGet-Pakete:

* Newtonsoft.Json: JSON-Serialisierung und -Deserialisierung.
* EntityFrameworkCore: ORM-Framework für Datenbankzugriffe.
* NUnit: Unit-Test-Framework.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| Spart Zeit und Aufwand | Abhängigkeiten zwischen Paketen |
| Viele Pakete verfügbar | Sicherheitsprobleme möglich |
| Leichte Integration | Updates können Kompatibilität brechen |
| Zentraler Zugriff (NuGet) | Eventuell Lizenzprobleme |

**Beispiel**:

2 Möglichkeiten der Installation eines NuGet-Paketes Ihrer Wahl. Z.B. Newtonsoft.Json:

**Newtonsoft.Json installieren**:

* Öffnen Sie in Visual Studio den **NuGet-Paket-Manager**:
  + Rechtsklick auf das Projekt > **NuGet-Pakete verwalten...**.
* Suchen Sie nach Newtonsoft.Json und installieren Sie es.

**Alternative: Installation über die CLI**:

* Öffnen Sie die **Paket-Manager-Konsole**:
  + Menüleiste > **Extras** > **NuGet-Paket-Manager** > **Paket-Manager-Konsole**.

Geben Sie ein, um ein Package zu installieren  
Install-Package Newtonsoft.Json

Bzw. um ein Package zu deinstallieren

uninstall-Package Newtonsoft.Json

**Und jetzt wenden Sie das installierte Paket an:**

using Newtonsoft.Json;

var person = new Person { Name = "Max", Alter = 25 };

string json = JsonConvert.SerializeObject(person);

Console.WriteLine(json);

class Person

{

public string Name { get; set; }

public int Alter { get; set; }

}

**Übung**

**1. Deserialisieren** Sie mitNewtonsoft.Json den JSON-String zurück in ein Objekt von Typ Person

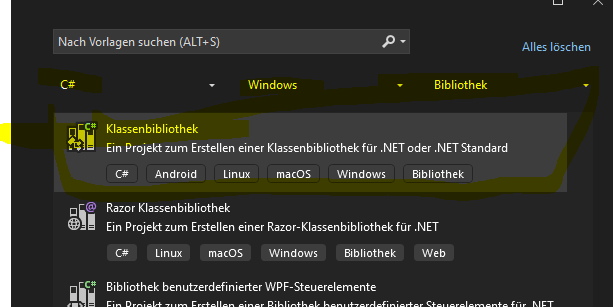
**2.** Suchen Sie im NugetPaket-Manager nach Komponente **SeriLog**. Installieren und deinstallieren Sie diese mal mit NugetPaket-Manager, mal in der Paket-Manager-Console. Wenden Sie die Komponente in Ihrem Programm an.

**3. Eigene Klassenbibliotheken (Dlls, Assemblys) herstellen**

* **Problem**: Häufig wiederkehrender Code in mehreren Projekten (z. B. Methoden zur Berechnung von Steuern, Konvertierungen von Formaten etc.)
* **Lösung**: Klassenbibliotheken (z. B. DLL-Dateien) ermöglichen **Wiederverwendbarkeit** und **Modularität**.

**Beispiel**

Erstellung eigener Klassenbibliothek



**Klasse im Projekt definieren:**

namespace MatheBibliothek

{

public class Rechner

{

public static int Addiere(int a, int b)

{

return a + b;

}

public static int Subtrahiere(int a, int b)

{

return a - b;

}

}

}

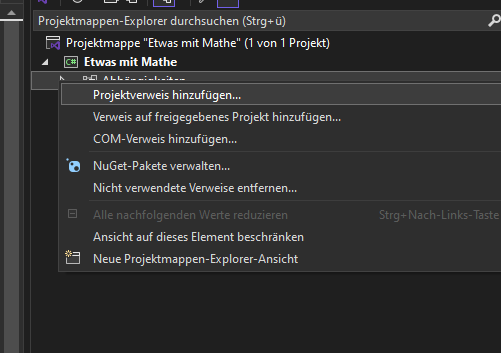
**Kompilieren der Bibliothek als Release** (Erstellen der .dll-Datei)

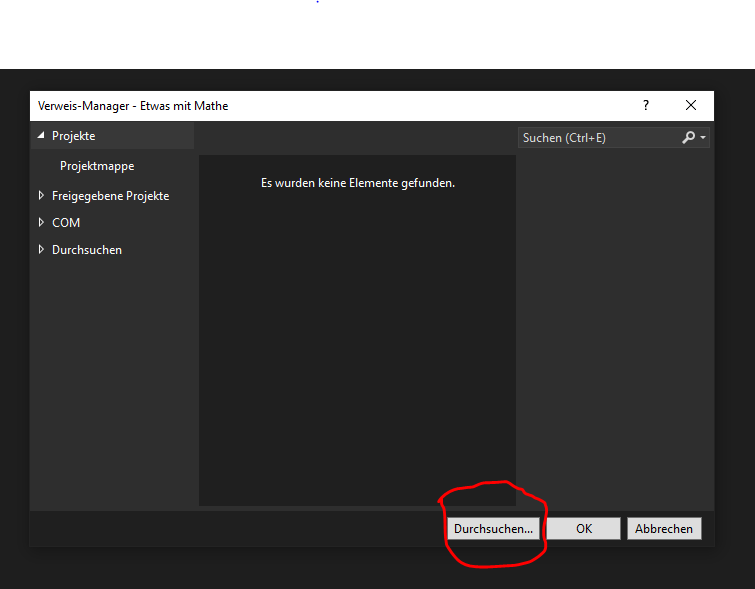
**Nutzung in einem Projekt**

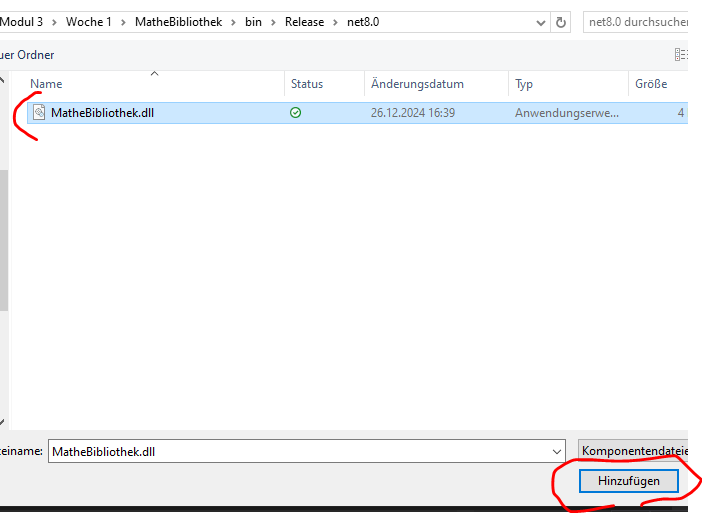
Neues Konsolenprojekt erstellen, **Referenz zur DLL hinzufügen** (Rechtsklick → Verweis hinzufügen)

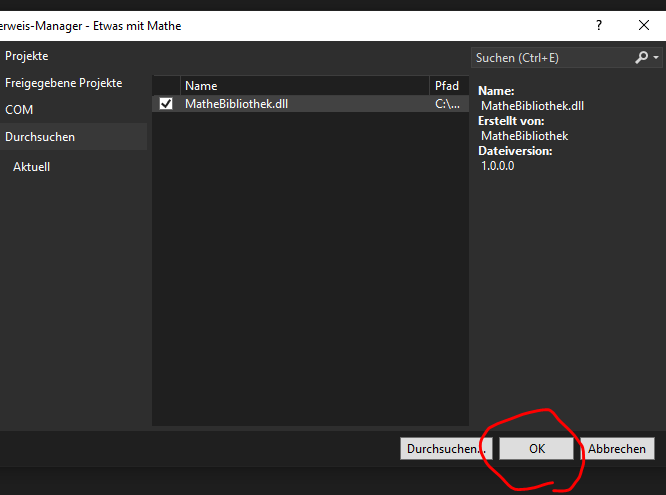
**Neue Konsolen-Anwendung,**

dann:









**Program.cs:**

using MatheBibliothek;

int summe = Rechner.Addiere(3, 5);

Console.WriteLine("Summe: " + summe);

**Übung**

Erstellen Sie ein neues Projekt vom Typ „Klassenbibliothek“ und benennen Sie es WetterBibliothek.

Erstellen Sie in WetterBibliothek eine Klasse WetterPrognose. Die Klasse soll eine Methode anbieten **GetWeatherCondition(int temperature),** die anhand des Parameters „Temperatur“ eine Wetterlage (wie: eiskalt, kalt, mild, warm, heiß) als String zurückliefert. In welchen Temperaturbereichen die jeweilige Wetterlage eintritt, bleibt es Ihnen überlassen.

Fügen Sie die erstellte WeatherLibrary.dll einer neuen Konsolen-Anwendung als Referenz hinzu und konsumieren Sie es, indem Sie GetWeatherCondition – Methode unter unterschiedlichen Temperaturen aufrufen.

**SOAP-Webservices aus dem Web**

Merkmale:

1. Verwendet XML:  
   Alle Nachrichten (Anfragen und Antworten) sind in XML strukturiert.
2. Plattform- und sprachunabhängig:  
   SOAP-Webservices können zwischen Systemen kommunizieren, die in unterschiedlichen Sprachen (z. B. Java, C#, PHP) geschrieben sind.
3. Transportprotokoll-unabhängig:  
   SOAP funktioniert primär über HTTP, kann aber auch andere Protokolle wie SMTP nutzen.
4. Verwendet WSDL (Web Services Description Language):  
   Eine WSDL-Datei beschreibt genau, welche Funktionen ein SOAP-Webservice anbietet, welche Parameter nötig sind und wie die Antworten aussehen.
5. Strikte Standardisierung:  
   SOAP folgt einem festen Schema und erlaubt durch Erweiterungen auch Sicherheit, Transaktionen und mehr (z. B. WS-Security).

Beispiel: **Country Info Service**

<http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL>

Der Country Info Service ist ein öffentlich zugänglicher **SOAP-Webservice**, der Informationen zu Ländern weltweit liefert. Er wurde als Beispiel-Service bereitgestellt, um zu zeigen, wie man SOAP-Webservices mit WSDL nutzen kann.

Wichtige Funktionen des Service

* CountryName(string ISOCode)  
  Gibt den vollständigen Namen des Landes zurück (z.B. "Germany" für "DE").
* ListOfCountryNamesByCode()  
  Liefert eine Liste aller Länder mit ISO-Code und Namen.
* CountryCurrency(string ISOCode)  
  Gibt Informationen zur Währung des Landes zurück.
* CountryFlag(string ISOCode)  
  Gibt die URL zur Flagge des Landes zurück.
* CapitalCity(string ISOCode)  
  Gibt die Hauptstadt des Landes zurück.

Wie funktioniert der Service?

* Du rufst eine Methode mit einem Parameter (meist ISO-Ländercode, z. B. "DE" für Deutschland) auf.
* Der Service antwortet mit den angefragten Daten (z. B. Namen, Hauptstadt).
* Die Kommunikation läuft über SOAP (XML-basierter Standard für Webservices).
* Die Struktur und Methoden sind in einer WSDL-Datei definiert, die Clients zur Code-Generierung verwenden.

**Einführung in REST** (Representational State Transfer)

**REST** ist ein Architekturstil für Webservices. Er definiert, wie Clients (z. B. ein C#-Programm oder ein Browser) über das Internet mit Servern kommunizieren.

REST verwendet das **HTTP-Protokoll**, so wie auch Webseiten – aber statt HTML liefern REST-Services typischerweise **Daten im JSON-Format**.

**🔑 Grundprinzipien von REST**

Ein REST-Service bietet den Zugriff auf **Ressourcen** – das können z. B. Benutzer, Produkte, Wetterdaten, Beiträge usw. sein.

Jede Ressource ist über eine eindeutige URL erreichbar. Man verwendet dabei unterschiedliche **HTTP-Methoden**, um etwas mit diesen Ressourcen zu tun:

|  |  |  |
| --- | --- | --- |
| Aktion | HTTP-Methode | Beschreibung |
| Lesen | GET | Daten abrufen (z. B. alle Benutzer) |
| Erstellen | POST | Neue Ressource anlegen |
| Aktualisieren | PUT | Vorhandene Ressource vollständig ersetzen |
| Teilweise ändern | PATCH | Ressource teilweise ändern |
| Löschen | DELETE | Ressource löschen |

**Beispiel: Benutzer-API**

Angenommen, ein REST-Service verwaltet Benutzer. Dann könnten folgende Aufrufe möglich sein:

| **URL** | **Methode** | **Zweck** |
| --- | --- | --- |
| /api/users | GET | Liste aller Benutzer abrufen |
| /api/users/5 | GET | Benutzer mit ID 5 abrufen |
| /api/users | POST | Neuen Benutzer erstellen |
| /api/users/5 | PUT | Benutzer 5 vollständig ersetzen |
| /api/users/5 | DELETE | Benutzer 5 löschen |

**REST arbeitet zustandslos**

Jeder Request enthält alle notwendigen Informationen. Der Server speichert **keinen Zustand** zwischen den Aufrufen – deshalb ist REST sehr skalierbar.

**Bekannte öffentliche REST-APIs**

|  |  |  |
| --- | --- | --- |
| Dienst | Beschreibung | Authentifizierung |
| JSONPlaceholder | Fake-API zum Testen (Posts, Users) | Nein |
| OpenWeatherMap | Wetterdaten weltweit | Ja (API-Key) |
| GitHub API | Infos zu Repos, Issues, Users | Optional |
| REST Countries | Infos zu Ländern weltweit | Nein |