Basics of Programming

Laboratory work 3.1

EXPLORING WORKING WITH FILES AND DESIGNING ENTITY RELATIONSHIPS

Variant - 5

*Prepared by:*

*student of SE-224B,*

*Demchuk Oksana*

*Accepted by:*

*Vasilieva M.D.*

**A diagram of a computer

Description automatically generatedTask**

1. Explore the structure of streaming I/O classes for working with files in C#.

2. Design the functionality in the form of a class diagram according to the variant.

3. Develop a C# program that meets the requirements in the task and variant. To demonstrate the operation, use text messages on the console through I/O operations. The program must demonstrate the application of streaming I/O classes to work with files writing, reading, and manipulating the data of the designed entities and the relationships between them.

4. Diagram(s) and source code must comply with basic design principles: OOP, composition over inheritance, loose coupling – high cohesion, inversion of control (IoC).

5. To receive points corresponding to "satisfactory", it is necessary to implement the task, according to the variant. File, I/O operations should be in separate classes, not in classes of business entities.

6. In order to receive points corresponding to "good", everything that is "satisfactory" must be implemented. And:

a. Allocate read-write operations to a file in a separate project of the solution. There should be no data manipulation or console I/O. There will be at least 2 projects in the solution

b. Screen input-output operations are in a separate ConsoleMenu class.

c. It is also necessary to use abstractions (abstract classes and interfaces) to demonstrate the relationships between entities in the variant.

7. For excellent mark, design entity relationships in such a way that other related types (such as Pupil or Musician) can be easily added without having to modify existing ones. And also – a new behavior. For example, when the Play() operation can be performed by a musician and a student.

8. \*The task of increased complexity is to work with one single data source file, writing and reading data of various types and entities.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Class Student elements** | **Operations with the students** | **Additional entities** | **Additional skills** |
| 5 | Surname, Course, Student ID, GPA, Country, number of the score book | Calculate the number of 3rd  year students living in  Ukraine. Get their data from  the file | McdonaldsWorker,  Manager | Play in chess |

**Code – data access level project**

**Entities.cs**

using database;

using System.Text.Json.Serialization;

namespace database

{

public interface IEntity

{

public string LastName { get; set; }

public string[] Methods { get; }

public string ToString();

}

public class Entity : IEntity, IPLayChess

{

string lastName;

public string LastName { get => lastName; set => lastName = value; }

[JsonConstructor]

public Entity(string LastNameInput)

{

lastName = LastNameInput;

}

public virtual string[] Methods { get { return new string[] {"PlayChess"}; } }

public string PlayChess()

{

return LastName + " plays chess";

}

public override string ToString() => lastName;

}

public class Student : Entity, IStudy

{

private string studentID;

private int? gpa;

private int? course;

private string? country;

private string? numberOfTheScoreBook;

public int? Course { get => course; set => course = value; }

public string StudentID { get => studentID; set => studentID = value; }

public int? GPA { get => gpa; set => gpa = value; }

public string? Country { get => country; set => country = value; }

public string? NumberOfTheScoreBook { get => numberOfTheScoreBook; set => numberOfTheScoreBook = value; }

public Student(string LastNameInput, string StudentIDInput) : base(LastNameInput)

{

studentID = StudentIDInput;

}

[JsonConstructor]

public Student(int? Course, string StudentID, int? GPA, string? Country, string? NumberOfTheScoreBook, string LastName) : base(LastName)

{

(studentID, gpa, course, country, numberOfTheScoreBook) = (StudentID, GPA, Course, Country, NumberOfTheScoreBook);

}

public Student(string LastName, string StudentID, int? Course, int? GPA, string? Country, string? NumberOfTheScoreBook) :

this(Course, StudentID, GPA, Country, NumberOfTheScoreBook, LastName)

{ }

public string Study()

{

course = course == 6 ? 1 : course + 1;

return LastName + " is now studing in " + course + " course";

}

public override string[] Methods { get { return base.Methods.Union(new string[] { "Study" }).ToArray(); } }

public override string ToString() =>

"Student - " + LastName +

", StudentID: " + studentID +

", Course: " + Course +

", GPA: " + GPA +

", Country: " + Country +

", NumberOfTheScoreBook: " + NumberOfTheScoreBook;

}

public class McWorker : Entity, ICook

{

public McWorker(string LastName) : base(LastName){}

public string Cook()

{

return LastName + " prepared your order!";

}

public override string[] Methods { get { return base.Methods.Union(new string[] { "Cook" }).ToArray(); } }

public override string ToString() => "McWorker - " + LastName;

}

public class Manager : Entity, IManage

{

public Manager(string LastName) : base(LastName) { }

public string Manage()

{

return LastName + " manages something!";

}

public override string[] Methods { get { return base.Methods.Union(new string[] { "Manage" }).ToArray(); } }

public override string ToString() => "Manager - " + LastName;

}

}

**Node.cs**

namespace database

{

public class Node<T>

{

public T data;

public Node<T>? next;

public Node<T>? prev;

public Node(T data)

{

this.data = data;

}

public Node(Node<T> data)

{

this.data = data.data;

}

}

}

**BidirectionalList.cs**

using System.Text;

namespace database

{

public class BidirectionalList<T>

{

private Node<T> firstNode;

private Node<T> lastNode;

private int size = 0;

public int Length => size;

public BidirectionalList() { }

public BidirectionalList(T input) => Push(input);

public T First => GetValue(0);

public T End => lastNode.data;

public void Push(T input) => Insert(size, input);

public void Insert(int index, T input)

{

CheckIndexValid(index, size + 1);

Node<T> newNode = new(input);

if (size == 0)

{

firstNode = newNode;

lastNode = newNode;

size++;

return;

}

if (index == 0)

{

newNode.next = firstNode;

firstNode.prev = newNode;

firstNode = newNode;

size++;

return;

}

if (index == size)

{

newNode.prev = lastNode;

lastNode.next = newNode;

lastNode = newNode;

size++;

return;

}

Node<T> cur = firstNode;

int currentIndex = 0;

while (cur.next != null && currentIndex + 1 != index)

{

cur = cur.next;

currentIndex++;

}

newNode.prev = cur;

newNode.next = cur.next;

if (cur.next != null) cur.next.prev = newNode;

cur.next = newNode;

if (index == size) lastNode = newNode;

size++;

}

public T Delete(int index)

{

CheckIndexValid(index, size);

Node<T> cur = firstNode;

if (index == 0)

{

firstNode = firstNode.next;

size -= 1;

return cur.data;

}

int currentIndex = 0;

while (cur.next != null && currentIndex != index)

{

cur = cur.next;

currentIndex++;

}

if (cur.prev != null) cur.prev.next = cur.next;

if (cur.next != null) cur.next.prev = cur.prev;

size -= 1;

return cur.data;

}

public T Pop() => Delete(size - 1);

public T GetValue(int index)

{

CheckIndexValid(index, size);

Node<T>? cur;

if (size / 2 >= index)

{

cur = firstNode;

int currentIndex = 0;

while (cur.next != null && currentIndex != index)

{

cur = cur.next;

currentIndex++;

}

}

else

{

cur = lastNode;

int currentIndex = size - 1;

while (cur.prev != null && currentIndex != index)

{

cur = cur.prev;

currentIndex--;

}

}

return cur.data;

}

public void SetValue(int index, T input)

{

CheckIndexValid(index, size);

Node<T> cur = firstNode;

int currentIndex = 0;

while (cur.next != null && currentIndex != index)

{

cur = cur.next;

currentIndex++;

}

cur.data = input;

}

public static void CheckIndexValid(int index, int size)

{

if (index < 0 || index >= size) throw new ArgumentOutOfRangeException("Index out of range");

}

public override String ToString()

{

StringBuilder toReturn = new("Total entities: " + size);

Node<T> cur = firstNode;

int currentIndex = 0;

while (cur != null)

{

toReturn.Append("\n" + (currentIndex + 1) + ". " + cur.data.ToString());

cur = cur.next;

currentIndex++;

}

return toReturn.ToString();

}

public T this[int position]

{

get => GetValue(position);

set => SetValue(position, value);

}

}

} **DataBase.cs**

using System.Text;

using System.Text.Json;

namespace database

{

public class DataBase

{

public readonly static string fileName = AppDomain.CurrentDomain.BaseDirectory + "database.txt";

public DataBase() { using StreamWriter w = File.AppendText(fileName); }

public BidirectionalList<Entity> Load()

{

BidirectionalList<Entity> list = new();

using (FileStream fileStream = File.OpenRead(fileName))

using (StreamReader streamReader = new StreamReader(fileStream, Encoding.UTF8, true, 256)) //Initializes a new instance of the StreamReader class for the specified file name, with the specified character encoding, byte order mark detection option, and buffer size.

{

String className = "Student";

String line;

int lineIndex = 0;

while ((line = streamReader.ReadLine()) != null)

{

if (lineIndex % 2 == 0)

{

className = line.Split(' ')[0];

}

else

{

Type EntityType = Type.GetType("database." + className) ?? throw new Exception("DataBase has unknown entity: " + className);

Entity element = (Entity)JsonSerializer.Deserialize(line, EntityType, new JsonSerializerOptions(JsonSerializerDefaults.Web));

list.Push(element);

}

lineIndex++;

}

}

return list;

}

public void Save(BidirectionalList<Entity> listToSave)

{

using StreamWriter writer = new(fileName);

int entityIndex = 0;

while (entityIndex < listToSave.Length)

{

writer.WriteLine(listToSave[entityIndex].GetType().Name + " " + listToSave[entityIndex].LastName);

writer.WriteLine(JsonSerializer.Serialize((object)listToSave[entityIndex]));

entityIndex++;

}

}

}

}

**Interfaces.cs**

namespace database

{

interface IStudy

{

public string Study();

}

interface IManage

{

public string Manage();

}

interface ICook

{

public string Cook();

}

interface IPLayChess

{

public string PlayChess();

}

}

**Code – Business logic level project**

**PeopleList.cs**

using database;

using System.Text.RegularExpressions;

namespace logic

{

public class PeopleList

{

BidirectionalList<Entity> data = new();

DataBase database = new();

public PeopleList()

{

data = database.Load();

}

public void Insert(Entity input)

{

data.Push(input);

database.Save(data);

}

public void Update(Entity input, int index)

{

data[index] = input;

database.Save(data);

}

public void Delete(int index)

{

data.Delete(index);

database.Save(data);

}

public void Save() => database.Save(data);

public int Length() => data.Length;

public static void ValidateName(string? name)

{

if (name == null || !Regex.Match(name, @"^\p{L}{1,32}$", RegexOptions.IgnoreCase).Success) throw new ArgumentException();

}

public static void ValidateID(string? id)

{

if (id == null || !Regex.Match(id, @"^KB\d{8}$", RegexOptions.IgnoreCase).Success) throw new ArgumentException();

}

public static void ValidateCourse(int? course)

{

if (course < 1 || course > 6) throw new ArgumentException();

}

public static void ValidateMark(int? mark)

{

if (mark < 0 || mark > 5) throw new ArgumentException();

}

public static void ValidateCountry(string? country)

{

if (country == null || !Regex.Match(country, @"^(\p{L}| |-){1,32}$", RegexOptions.IgnoreCase).Success) throw new ArgumentException();

}

public static void ValidateNumberOfTheScoreBook(string? number)

{

if (number == null || !Regex.Match(number, @"^\d{6}$", RegexOptions.IgnoreCase).Success) throw new ArgumentException();

}

public List<Tuple<int, Student>> Search()

{

List<Tuple<int, Student>> Entities = new();

if (data.Length == 0) { return Entities; }

for (int i = 0; i < data.Length; i++)

{

Entity cur = data[i];

if (cur is Student)

{

Student student = cur as Student;

if (student is not null)

{

if ((student.Country == "Ukraine" || student.Country == "ukraine") && student.Course == 3) Entities.Add(Tuple.Create(i, student));

}

}

}

return Entities;

}

public Entity this[int position]

{

get => data[position];

set => data.SetValue(position, value);

}

}

}

**Code – Presentation level project**

**using database;**

**using logic;**

**using System.Reflection;**

**namespace console**

**{**

**internal class Program**

**{**

**PeopleList list = new();**

**public void ShowPeople(string? typeOfEntity)**

**{**

**Console.Clear();**

**Console.WriteLine(list.Length() + " entities:");**

**for (int index = 0; index < list.Length(); index++)**

**{**

**if (typeOfEntity == null || list[index].GetType().Name == typeOfEntity)**

**{**

**Console.WriteLine(1 + index + ". " + list[index]);**

**}**

**}**

**Console.WriteLine("0 - back to menu, entity number - view entity");**

**bool done = false;**

**do**

**{**

**var i = Console.ReadLine();**

**switch (i)**

**{**

**case "0": ShowMain(); return;**

**default:**

**try**

**{**

**int parsed = Convert.ToInt32(i);**

**if (parsed < 1 || parsed > list.Length()) throw new ArgumentException();**

**ViewEntity(parsed - 1);**

**done = true;**

**return;**

**}**

**catch (Exception) { Console.WriteLine("Invalid input!"); }**

**break;**

**}**

**} while (!done);**

**}**

**public void ViewEntity(int index)**

**{**

**Console.Clear();**

**Console.WriteLine("Entity: " + (index + 1) + " of " + list.Length() + " items");**

**Entity entity = list[index];**

**Console.WriteLine(1 + index + ". " + entity);**

**Console.WriteLine("This entity can:");**

**for (int i = 0; i < entity.Methods.Length; i++)**

**{**

**Console.WriteLine(entity.Methods[i] + " ");**

**}**

**Console.WriteLine("0 - go back to menu; 1 - edit; 2 - delete; do something;");**

**bool done = false;**

**do**

**{**

**var input = Console.ReadLine();**

**if (input != null && input.StartsWith("do "))**

**{**

**String MethodName = input.Split(' ')[1]; ;**

**Type type = entity.GetType();**

**MethodInfo? theMethod = type.GetMethod(MethodName);**

**if (theMethod == null)**

**{**

**Console.WriteLine("Error: no such function");**

**}**

**else**

**{**

**Console.WriteLine(theMethod.Invoke(entity, new string[] { }));**

**list.Save();**

**}**

**}**

**else**

**{**

**switch (input)**

**{**

**case "0": ShowMain(); return;**

**case "1": AddOrEdit(entity, index, null); return;**

**case "2":**

**try**

**{**

**list.Delete(index);**

**ShowMain(); return;**

**}**

**catch (Exception) { Console.WriteLine("Failed to delete!"); }**

**break;**

**default: Console.WriteLine("Invalid input!"); break;**

**}**

**}**

**}**

**while (!done);**

**}**

**public void AddOrEdit(Entity? entity, int? index, string? type)**

**{**

**Console.Clear();**

**bool isEditing = entity != null;**

**Console.WriteLine(isEditing ? "Editing " + entity.GetType().Name : "Adding " + type);**

**if (isEditing)**

**{**

**Console.WriteLine(entity.ToString());**

**type = entity.GetType().Name;**

**}**

**Entity? newEntity = null;**

**switch (type)**

**{**

**case "Student":**

**Student? oldStudent = isEditing ? (Student)entity : null;**

**newEntity = new Student(**

**AskName(isEditing ? oldStudent.LastName : null),**

**AskID(isEditing ? oldStudent.StudentID : null),**

**AskCourse(isEditing ? oldStudent.Course : null),**

**AskGPA(isEditing ? oldStudent.GPA : null),**

**AskCountry(isEditing ? oldStudent.Country : null),**

**AskNumberOfTheScoreBook(isEditing ? oldStudent.NumberOfTheScoreBook : null)**

**);**

**ShowMain();**

**break;**

**case "McWorker":**

**McWorker? oldTailor = isEditing ? (McWorker)entity : null;**

**newEntity = new McWorker(**

**AskName(isEditing ? oldTailor.LastName : null)**

**);**

**ShowMain();**

**break;**

**case "Manager":**

**Manager? oldSinger = isEditing ? (Manager)entity : null;**

**newEntity = new Manager(**

**AskName(isEditing ? oldSinger.LastName : null)**

**);**

**ShowMain();**

**break;**

**default:**

**Console.WriteLine("I don`t know how to create this entity(\nNew key to go back");**

**Console.ReadKey();**

**break;**

**}**

**if (newEntity != null)**

**{**

**if (isEditing)**

**{**

**list.Update(newEntity, (int)index);**

**}**

**else**

**{**

**list.Insert(newEntity);**

**}**

**}**

**return;**

**}**

**public string? AskName(string? input)**

**{**

**bool done = false;**

**string name = "";**

**do**

**{**

**Console.WriteLine("Enter name: " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "" && input != null) { return input; }**

**try**

**{**

**PeopleList.ValidateName(stringFromConsole);**

**name = stringFromConsole;**

**}**

**catch (Exception) { Console.WriteLine("Wrong name!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return name;**

**}**

**public string? AskID(string? input)**

**{**

**bool done = false;**

**string id = "";**

**do**

**{**

**Console.WriteLine("Enter id (like KB00000000): " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "" && input != null) { return input; }**

**try**

**{**

**PeopleList.ValidateID(stringFromConsole);**

**id = stringFromConsole;**

**}**

**catch (Exception) { Console.WriteLine("Wrong id!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return id;**

**}**

**public int? AskCourse(int? input)**

**{**

**bool done = false;**

**int course = 0;**

**do**

**{**

**Console.WriteLine("Enter course: " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "")**

**{**

**if (input != null) { return input; }**

**return null;**

**}**

**try**

**{**

**int parsed = Convert.ToInt32(stringFromConsole);**

**PeopleList.ValidateCourse(parsed);**

**course = parsed;**

**}**

**catch (Exception) { Console.WriteLine("Wrong course!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return course;**

**}**

**public int? AskGPA(int? input)**

**{**

**bool done = false;**

**int gpa = 0;**

**do**

**{**

**Console.WriteLine("Enter GPA: " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "")**

**{**

**if (input != null) { return input; }**

**return null;**

**}**

**try**

**{**

**int parsed = Convert.ToInt32(stringFromConsole);**

**PeopleList.ValidateMark(parsed);**

**gpa = parsed;**

**}**

**catch (Exception) { Console.WriteLine("Wrong gpa!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return gpa;**

**}**

**public string? AskCountry(string? input)**

**{**

**bool done = false;**

**string? country = null;**

**do**

**{**

**Console.WriteLine("Enter Country: " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "")**

**{**

**if (input != null) { return input; }**

**return null;**

**}**

**try**

**{**

**PeopleList.ValidateCountry(stringFromConsole);**

**country = stringFromConsole;**

**}**

**catch (Exception) { Console.WriteLine("Wrong Country!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return country;**

**}**

**public string? AskNumberOfTheScoreBook(string? input)**

**{**

**bool done = false;**

**string? numberOfTheScoreBook = null;**

**do**

**{**

**Console.WriteLine("Enter Number of the score book: " + (input != null ? "(" + input + ")" : ""));**

**string? stringFromConsole = Console.ReadLine();**

**if (stringFromConsole == null || stringFromConsole == "")**

**{**

**if (input != null) { return input; }**

**return null;**

**}**

**try**

**{**

**PeopleList.ValidateNumberOfTheScoreBook(stringFromConsole);**

**numberOfTheScoreBook = stringFromConsole;**

**}**

**catch (Exception) { Console.WriteLine("Wrong Number of the score book!"); continue; }**

**done = true;**

**}**

**while (!done);**

**return numberOfTheScoreBook;**

**}**

**public void Search()**

**{**

**List<Tuple<int, Student>> searchList = list.Search();**

**Console.Clear();**

**Console.WriteLine("Search: " + searchList.Count + " items ");**

**int currentIndex = 0;**

**while (currentIndex < searchList.Count)**

**{**

**Console.WriteLine(1 + searchList[currentIndex].Item1 + ". " + searchList[currentIndex].Item2);**

**currentIndex++;**

**}**

**Console.WriteLine("0 - menu");**

**bool done = false;**

**do**

**{**

**var input = Console.ReadLine();**

**switch (input)**

**{**

**case "0": ShowMain(); return;**

**default:**

**ShowMain(); return;**

**}**

**}**

**while (!done);**

**}**

**public void ShowMain()**

**{**

**Console.Clear();**

**Console.WriteLine(**

**"1 - view all;\n" +**

**"2 - view the database of students;\n" +**

**"3 - view the database of mcworkers;\n" +**

**"4 - view the database of managers;\n" +**

**"5 - add students to the database;\n" +**

**"6 - add mcworkers to the database;\n" +**

**"7 - add managers to the database;\n" +**

**"8 - search\n" +**

**"0 - EXIT");**

**}**

**static void Main(String[] args)**

**{**

**bool flag = false;**

**Program pr = new();**

**pr.ShowMain();**

**do**

**{**

**Console.WriteLine("Enter command:");**

**string? i = Console.ReadLine();**

**if (!int.TryParse(i, out int result)) { Console.WriteLine("Invalid input!"); }**

**else**

**{**

**int command = int.Parse(i);**

**switch (command)**

**{**

**case 1:**

**pr.ShowPeople(null);**

**break;**

**case 2:**

**pr.ShowPeople("Student");**

**break;**

**case 3:**

**pr.ShowPeople("McWorker");**

**break;**

**case 4:**

**pr.ShowPeople("Manager");**

**break;**

**case 5:**

**pr.AddOrEdit(null, null, "Student");**

**break;**

**case 6:**

**pr.AddOrEdit(null, null, "McWorker");**

**break;**

**case 7:**

**pr.AddOrEdit(null, null, "Manager");**

**break;**

**case 8:**

**pr.Search();**

**break;**

**case 0:**

**flag = true;**

**break;**

**default:**

**Console.WriteLine("Invalid number!");**

**break;**

**}**

**}**

**} while (!flag);**

**}**

**}**

**}**

**Conclusion**

During the laboratory work we studied working with files using streams in the C# language using various related entities.

**Self-checking questions**

**1. What is a stream?**

In C#, a stream is a sequence of data elements that can be read from or written to sequentially. Streams are a fundamental concept in C# for working with input and output operations, and they provide an abstraction for reading and writing data from various sources, such as files, network connections, memory buffers, and more. Streams provide a consistent way to handle data regardless of its source or destination.

**2. Describe the stream input/output class hierarchy in C#.**

In C#, the stream input/output class hierarchy is organized within the System.IO namespace and follows a hierarchy of abstract and concrete classes. This hierarchy provides a common interface and base classes for working with different types of data streams. Below is an overview of the key classes in the stream input/output class hierarchy:

Stream (Abstract Base Class):

Stream is the abstract base class for all stream classes in C#.

It provides a common set of methods and properties for reading and writing data.

Derived classes implement specific behaviors for different types of streams.

Derived Stream Classes:

There are several derived stream classes that provide specific implementations for different data sources or destinations. Some of the commonly used ones include:

FileStream: Used for reading from and writing to files on disk.

MemoryStream: Allows working with an in-memory buffer as a stream.

NetworkStream: Used for reading and writing data over network connections.

CryptoStream: Wraps another stream and provides cryptographic operations on the data.

GZipStream and DeflateStream: Compress or decompress data while reading or writing.

BufferedStream: Adds buffering functionality to improve I/O performance.

PipeStream: Used for interprocess communication (IPC) between threads or processes.

Many others, depending on specific use cases.

Text Stream Classes:

There are specialized stream classes for working with text data, which include:

StreamReader: Reads text from a stream using a specified encoding.

StreamWriter: Writes text to a stream using a specified encoding.

StringReader and StringWriter: Work with strings as if they were streams.

Binary Stream Classes:

These classes are designed for working with binary data and include:

BinaryReader: Reads binary data from a stream, allowing you to read primitive data types.

BinaryWriter: Writes binary data to a stream, allowing you to write primitive data types.

**3. List the read/write methods for FileStream in C#.**

Reading Methods:

Read(byte[] buffer, int offset, int count): Reads a specified number of bytes from the stream into the provided byte array starting at the given offset.

ReadByte(): Reads a single byte from the stream and returns it as an integer (or -1 if the end of the stream is reached).

ReadAsync(byte[] buffer, int offset, int count): Asynchronous version of Read() for non-blocking I/O operations.

CopyTo(Stream destination): Reads all the bytes from the current stream and writes them to the destination stream.

Writing Methods:

Write(byte[] buffer, int offset, int count): Writes a specified number of bytes from the provided byte array to the stream, starting at the given offset.

WriteByte(byte value): Writes a single byte to the stream.

WriteAsync(byte[] buffer, int offset, int count): Asynchronous version of Write() for non-blocking I/O operations.

Flush(): Flushes the stream, ensuring that any buffered data is written to the file immediately.

**4. Explain how to create a character stream in C#.**

using System;

using System.IO;

class Program

{

static void Main()

{

// Create a StreamReader to read from a file (you can also use other sources)

using (StreamReader reader = new StreamReader("example.txt"))

{

string line;

while ((line = reader.ReadLine()) != null)

{

Console.WriteLine(line); // Process the text data

}

}

}

}

**5. Explain how thread redirection is done in C#.**

using System;

using System.IO;

class Program

{

static void Main()

{

// Create a StreamWriter to write to a file

using (StreamWriter fileStream = new StreamWriter("output.txt"))

{

// Redirect the standard output to the file stream

Console.SetOut(fileStream);

// Now, anything written to Console.Out will go to the file

Console.WriteLine("This will be written to output.txt");

} // The StreamWriter will be disposed, restoring the original Console.Out

// Now, standard output is back to the console

Console.WriteLine("This will be displayed in the console.");

}

}

**6. Give examples of methods for reading/writing values of predefined types to binary streams in C#.**

Writing to a Binary Stream (BinaryWriter):

using System;

using System.IO;

class Program

{

static void Main()

{

using (FileStream fileStream = new FileStream("data.dat", FileMode.Create))

{

using (BinaryWriter writer = new BinaryWriter(fileStream))

{

int intValue = 42;

double doubleValue = 3.14;

string stringValue = "Hello, Binary!";

writer.Write(intValue);

writer.Write(doubleValue);

writer.Write(stringValue);

}

}

}

}

Reading from a Binary Stream (BinaryReader):

using System;

using System.IO;

class Program

{

static void Main()

{

using (FileStream fileStream = new FileStream("data.dat", FileMode.Open))

{

using (BinaryReader reader = new BinaryReader(fileStream))

{

int intValue = reader.ReadInt32();

double doubleValue = reader.ReadDouble();

string stringValue = reader.ReadString();

Console.WriteLine($"Int: {intValue}, Double: {doubleValue}, String: {stringValue}");

}

}

}

}

**7. Explain using regular expressions.**

Regular expressions (regex or regexp) are powerful patterns used for searching, matching, and manipulating text. In C#, you can use the **System.Text.RegularExpressions** namespace to work with regular expressions.

**8. What is the difference between composition and generalization. Which type of connection and when should be used? Why?**

Composition: Composition is a design principle where one class contains another class as a part or component. It represents a "has-a" relationship. In composition, the contained class does not have a separate existence outside of the containing class. If the containing class is destroyed, the contained class is also destroyed.

Generalization (Inheritance): Generalization, often referred to as inheritance, is a design principle where a class inherits properties and behaviors from another class. It represents an "is-a" relationship. Inheritance allows a subclass to reuse and extend the attributes and methods of a superclass.

When to Use Composition:

Use composition when you want to create a relationship between two classes where one class is a part or component of the other, and the lifetime of the contained class is tied to the lifetime of the containing class.

Composition is useful when you want to build complex objects by combining simpler objects or when you want to encapsulate functionality within a class without exposing the internal details.

When to Use Generalization (Inheritance):

Use generalization (inheritance) when you have a clear "is-a" relationship between classes, and you want to model a hierarchy of related objects.

Inheritance is useful when you want to reuse code and behavior from a base class in one or more derived classes, promoting code reuse and maintaining a common interface.

Why Composition over Inheritance?:

Composition is often favored over inheritance because it provides more flexibility and avoids some of the issues associated with deep class hierarchies.

Inheritance can lead to tight coupling between classes, making it harder to change the behavior of a class without affecting its subclasses. Composition allows for greater decoupling and flexibility.

**9. What is the difference between an abstract class and an interface?**

Abstract Class:

An abstract class is a class that cannot be instantiated directly; it serves as a blueprint for other classes.

It can have both abstract (unimplemented) methods and concrete (implemented) methods.

An abstract class can contain fields, properties, and constructors.

A class can inherit from only one abstract class.

It supports the concept of code reusability through inheritance.

Abstract classes are often used when you want to provide a common base for a group of related classes.

Interface:

An interface is a contract that defines a set of method signatures (without implementations) that a class that implements the interface must provide.

Interfaces cannot have fields, properties, or constructors. They only define method signatures, properties, events, and indexers.

A class can implement multiple interfaces, enabling it to exhibit multiple behaviors.

Interfaces support the concept of multiple inheritance through implementation.

Interfaces are often used when you want to define a contract that multiple classes can adhere to, promoting code interoperability and enabling classes with different inheritance hierarchies to share common functionality.

**10. What is the benefit of using abstractions instead of concrete implementations? Give examples.**

Using abstractions instead of concrete implementations is a fundamental principle of object-oriented programming and software design. The main benefits include:

Flexibility: Abstractions allow you to change or extend the behavior of your code more easily without affecting the rest of the system. You can replace one implementation with another that adheres to the same abstraction without breaking the code that relies on it.

Code Reusability: Abstractions promote code reusability by defining common interfaces or base classes that can be implemented or extended by multiple concrete classes. This reduces code duplication and encourages a more efficient and maintainable codebase.

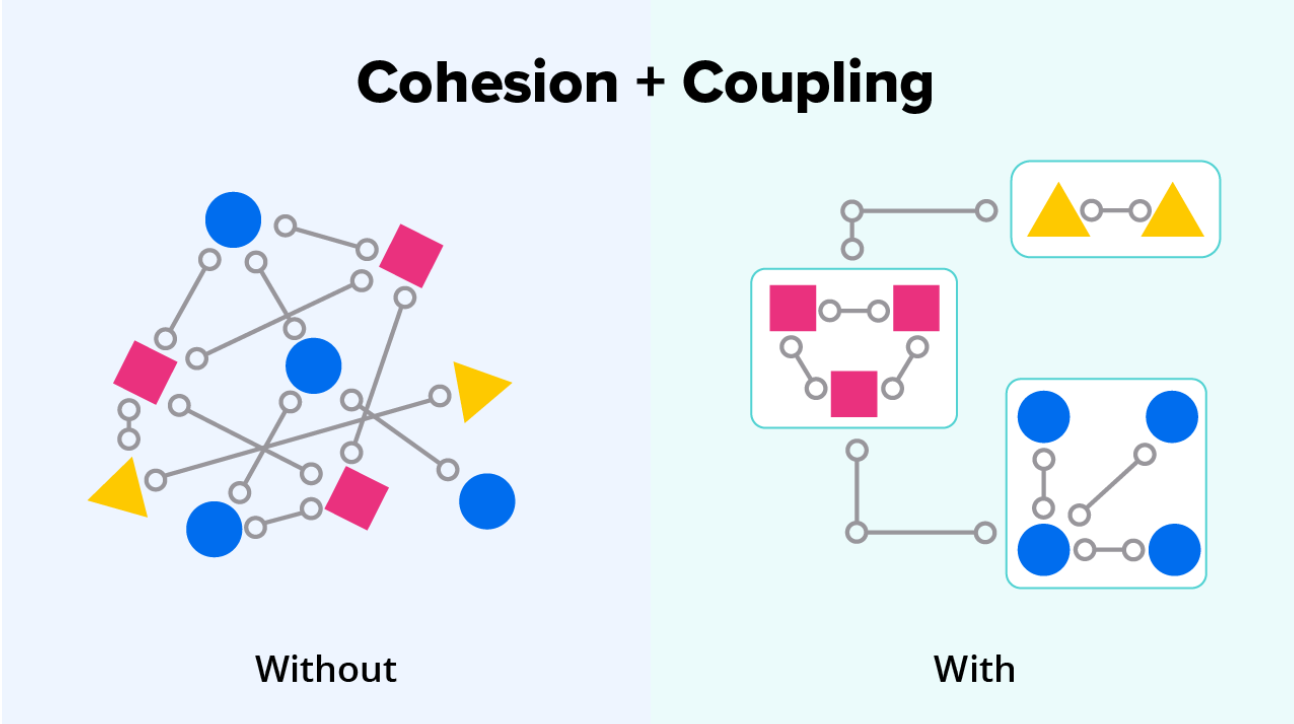
Testing: Abstractions make it easier to write unit tests and mock objects. You can create mock implementations of interfaces or abstract classes for testing purposes, isolating the code being tested from external dependencies.

Interoperability: Abstractions enable interoperability between different components or systems. If two systems adhere to a common interface or share an abstract contract, they can communicate and work together seamlessly.

**11. What is the principle of inversion of control (inversion of control – IoC)**

"Composition over inheritance" – is a principle in object-oriented programming (OOP) that indicates that in most cases it is better to use object composition (combining objects into a larger object) instead of inheriting functionality from other classes (inheritance).

**Loose Coupling:** This principle indicates that the classes and components of the application should minimize the dependence on other classes and components as much as possible. That is, classes should be as little dependent as possible on the implementation details of other classes, and changes in one class should not adversely affect other classes.

**High Cohesion:** This principle indicates that within one class there should be a high concentration of functionality that logically relates to each other. In other words, classes should perform only one responsibility or task, and that task should be well defined. 

**"Inversion of Control" (IoC)** is an application design principle that states that an object or

component should not independently control or create its dependencies or resources, but these

dependencies or resources should be provided to the object from the outside. This principle

makes applications more flexible, easier to test and maintain, and promotes separation of

components.