***Part one – JavaScript***

1.Output  
{  
outer func: this.name=bar

outer func: self.name=bar

inner func: this.name=undefined

inner func: self.name=bar

}  
Because

In func(), this points to obj because func is called by obj.func().

self is just a way to save this so we can use it later inside other functions.

In the outer part (inside func() but before the inner function), both this and self point to obj, so both this.fullName and self.fullName show "bar".

Inside the inner function (the IIFE), this doesn’t point to obj anymore. It points to window (or undefined if strict mode is on). So this.fullName is undefined.

But self still points to obj because it was saved earlier. So self.fullName is still "bar".

2.Output

{

5  
10  
}  
Because

functionList1 works because each time func(i) runs, it makes a new function, and that function remembers the i value at that time. So every function in functionList1 keeps its own i.

functionList2 is different. It doesn’t save i at all. All the functions in functionList2 just look at i directly from the loop. After the loop finishes, i is already 10. So all of them just see 10, no matter which one you call.

3.Output  
{  
value of x is:undefined  
}  
Because  
Inside the function, var x gets hoisted, meaning the declaration moves to the top but not the value. So when console.info() runs, it sees x as undefined because the assignment x = 18 hasn't happened yet. The global x doesn't matter here because the function has its own x.

***Part two – C#***

1.Output  
{

8

8

8

5

5

8

}  
Because  
Class A is a **Reference Type**, so changes affect all variables pointing to the same object.  
Class B is a **Value Type**, so each variable gets its own copy, and changes to one don’t affect the others.  
When passing a class to a method, we can modify the object inside the method, but if we create a **new object** inside, it breaks the link to the original.  
If a method returns a **new object** and we assign it, the old object reference gets replaced.

2.  
using System;

using System.Collections.Generic;

public static class EnumerableExtensions

{

public static void Loop<T>(this IEnumerable<T> source, Action<T> action)

{

if (source == null)

{

throw new ArgumentNullException(nameof(source), "Source collection cannot be null.");

}

if (action == null)

{

throw new ArgumentNullException(nameof(action), "Action cannot be null.");

}

foreach (T item in source)

{

action(item);

}

}

}  
Sample Usage  
using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

IEnumerable<string> list = new List<string>

{

"Value 1",

"Value 2"

};

list.Loop<string>(value => Console.WriteLine($"Output Value: {value}"));

}

}

3.  
private static readonly ConcurrentDictionary<Type, PropertyInfo[]> PropertyCache = new();

public Dictionary<string, object> Convert<T>(T t) where T : new()

{

if (t == null)

return new Dictionary<string, object>();

var result = new Dictionary<string, object>();

var properties = PropertyCache.GetOrAdd(typeof(T), type => type.GetProperties());

foreach (var prop in properties)

{

var value = prop.GetValue(t);

if (value is DateTime dateTimeValue)

{

result[prop.Name] = dateTimeValue.ToString("yyyy-MM-dd");

}

else

{

result[prop.Name] = value;

}

}

return result;

}

Logic Flow for Convert<T> Method:

1. Input Validation:
   1. Check if the input object t is null.
   2. If t is null, return an empty dictionary immediately.
2. Caching Property Information:
   1. Use reflection to get the properties of the type T.
   2. To optimize performance, cache the properties of each type in a static dictionary (e.g., ConcurrentDictionary<Type, PropertyInfo[]>).
   3. This ensures that reflection only occurs once per type, avoiding repeated expensive reflection calls on the same type.
3. Property Value Extraction:
   1. For each property of the object t, get the value of the property using reflection.
   2. If the property is a DateTime, format it as a string (e.g., "yyyy-MM-dd").
   3. Otherwise, store the property’s value as is.
4. Store in Dictionary:
   1. For each property, add a key-value pair to a new dictionary, where the key is the property name and the value is the property value.
5. Return Result:
   * Return the dictionary containing the property names as keys and their corresponding values.

4. **Ways to prevent uploading malicious files**

File Validation: Check file type by content, not just extension.

Whitelist Extensions: Only allow specific file types.

Antivirus Scan: Scan uploaded files for malware.

File Size Limit: Restrict file size.

Quarantine: Store uploads in an isolated area first.

Input Sanitization: Strip out dangerous characters in file names.

MIME Type Checking: Compare the MIME type against the file extension.

Log Uploads: Track all uploads.

Secure Storage: Store files outside the web root directory.

***Part three – SQL***

1.Logical Processing Order of SELECT Statement in SQL Server:

1. FROM – Retrieve data from the specified tables or views.
2. JOIN – Apply the join operations (e.g., INNER JOIN, LEFT JOIN) if present.
3. ON – Apply the join condition.
4. WHERE – Filter rows based on conditions.
5. GROUP BY – Group rows based on specified columns.
6. HAVING – Filter groups after aggregation.
7. SELECT – Choose the columns to be returned.
8. DISTINCT – Remove duplicate rows.
9. ORDER BY – Sort the result set.
10. TOP – Limit the number of rows returned.

2.

DECLARE @PageIndex INT = 0;

DECLARE @PageSize INT = 5;

SELECT \*

FROM (

SELECT

ROW\_NUMBER() OVER (ORDER BY StudentID) AS RowNum

StudentID,

StudentName,

Age,

Gender,

Grade

FROM dbo.Student

) AS StudentsWithRowNum

WHERE RowNum BETWEEN (@PageIndex \* @PageSize + 1) AND ((@PageIndex + 1) \* @PageSize);

3.

### Steps to Improve Database Performance and Recovery

Performance:

Indexes: Proper indexing on columns used in joins, filters, and sorting.

Partitioning: Split large tables for better performance.

Query Optimization: Avoid SELECT \*, use proper joins, filter early.

Connection Pooling: Use connection pools to reduce overhead.

Caching: Cache frequent queries.

Statistics Update: Keep table statistics up-to-date.

Use Stored Procedures: Precompiled execution plans.

Recovery:

Backup Strategy: Regular full, differential, and log backups.

Point-in-Time Recovery: Use log backups.

Redundancy: Use database mirroring or replication.

Monitoring: Set up alerts for high CPU, long queries, disk space.

Testing Recovery: Periodically test restoring from backups.